E.T.S de Ingeniería Industrial, Informática y de Telecomunicación

09/06/2015

Receptor ARINC 429

Grado en Ingeniería en Tecnologías de Telecomunicación

Aitor Esquíroz Olcoz

Directores de proyecto

- Orbital: Carlos Zubieta Arricivita
- UPNA: Santiago J. Led Ramos

Abstract

The aim of this project is to design and implement an ARINC 429 receiver with an AXI communication interface using a hardware description language (HDL). This includes the design and execution of all kinds of required simulations to implement a pre-certifiable IP Core, as well as programming and testing it on a FPGA. This is made by using Xilinx tools as Vivado and SDK for programming and using Xsim, Modelsim and Octave for executing and testing the simulations.

The design is approached in a modular way, breaking down the project into seven blocks and four phases of development. the blocks are the AXI slave, the FIFO, the Logic Control, the Line Receiver, the Latch, the Label Check and the Fifo To Output Register. The development phases are designing and implementing the blocks, designing and implementing the simulations, synthesizing and implementing, and programming and testing the FPGA.

Resumen

El objeto de este proyecto es diseñar e implementar un receptor ARINC 429 con un interfaz de comunicación AXI usando un lenguaje de descripción hardware. También abarca el diseño y ejecución de todos los tipos de simulación requeridos para hacer un IP Core pre-certificable, así como la programación y testeo en la FPGA. Se realiza con las herramientas de Xilinx Vivado y SDK para la programación y con Xsim, Modelsim y Octave para las simulaciones.

El diseño se aborda de forma modular, descomponiendo el proyecto en siete bloques y cuatro fases de desarrollo. Los bloques son el esclavo AXI, la FIFO, la lógica de control, el receptor de linea, el volteador de etiquetas (latch) , el comprobador de etiquetas y el registro entre fifo y salida. Y las fases de desarrollo son el diseño e implementación de los bloques, el diseño e implementación de las simulaciones, la síntesis e implementación, y la programación en la FPGA.

Palabras clave

ARINC 429, Aviónica, AXI, FPGA, IP Core, Modelsim, Octave, Script, Receptor, VHDL, Vivado, Xilinx, Xsim, Zedboard, Zynq.

Índice de contenidos

1. Introducción 5
2. Objetivos 6
3. Metodología 8
3.1 Implementación de Módulos y Testbenchs 8
3.2 Agilismo Aplicado al Proyecto 11
3.3 RTCA/DO 254 12
4. Desarrollo e Implementación 13
4.1 Introducción 13
4.2 Definición Técnica de un Receptor ARINC 429 13
4.3 Definición concreta del interfaz esclavo AXI 16
4.4 Uso del IP Core 19
4.5 Funcionamiento de los Módulos del Receptor 20
4.5.1 Line Receiver 20
4.5.2 Latch 22
4.5.3 Label Check 23
4.5.4 Fifo To Output Register 24
4.5.5 Logic Control 25
4.5.6 FIFO 26
4.5.7 Esclavo AXI 27
5. Diseño e Implementación de los Testbenches 30
5.1 Testbench de los Módulos del Receptor 30
5.1.1 Testbench del Line Receiver 30
5.1.2 Testbench del Latch 32
5.1.3 Testbench del Label Check 33
5.1.4 Testbench del Fifo To Output Register 34
5.1.5 Testbench del Logic Control 35
5.2 Testbench del Receptor Completo 36
6. Síntesis e Implementación 38
7. Programación de la FPGA 39
8. Análisis de Resultados y Conclusiones 42
9. Líneas Futuras 44
10. Bibliografía 45

Tabla de Ilustraciones

Ilustración 1. Formato de palabra en ARINC 429 5
Ilustración 2. Diseño preliminar del receptor A429 con los módulos a implementar 7
Ilustración 3. Estructura tipo del método de diseño de dos procesos..... 9
Ilustración 4. Estructura tipo del método empleado para realizar testbenches automáticos ... 10
Ilustración 5. Definición gráfica de la distribución del tiempo en un proyecto llevado con Scrum (4) 11
Ilustración 6. Diseño preliminar con interconexiones entre registros del AXI y los módulos del receptor 14
Ilustración 7. GUI de configuración del módulo FIFO Generator 14
Ilustración 8. Gráfico de pulso para definición de las tolerancias de la línea de salida (5) 15
Ilustración 9. GUI de generación de esclavo AXI básico..... 16
Ilustración 10. Receptor ARINC 429 completo 19
Ilustración 11. Esquemático del módulo Line Receiver 20
Ilustración 12. Formato de recepción de palabra en A429 en sentido de izquierda a derecha. 21
Ilustración 13. Formato en A429 de almacenamiento de palabra en registro de salida. 21
Ilustración 14. Esquemático de módulo Latch 22
Ilustración 15. Esquemático del módulo Label Check..... 23
Ilustración 16. Mapeo de etiquetas 23
Ilustración 17. Esquemático del módulo Fifo To Output Register 24
Ilustración 18. Esquemático del módulo Logic Control..... 25
Ilustración 19. Visualización del funcionamiento de la FIFO (6) 26
Ilustración 21. Selección del nombre y path del AXI esclavo 28
Ilustración 20. Módulo FIFO en formato caja negra 27
Ilustración 22. Definición del tipo de interfaz AXI a generar 28
Ilustración 23. Módulo esclavo AXI personalizado en formato caja negra 29
Ilustración 24. Interconexión interna entre registros y salidas adicionales..... 30
Ilustración 25. Simulación del testbench automático del módulo Line Receiver 31
Ilustración 26. Simulación del testbench automático del módulo Latch 32
Ilustración 27. Simulación del testbench automático del módulo Label Check 33
Ilustración 28. Simulación del testbench automático del módulo Fifo To Output Register 35
Ilustración 29. Simulación del testbench automático del módulo Logic Control 36
Ilustración 30. Simulación del testbench automático del Receptor completo 37
Ilustración 31. Esquemático de un módulo sintetizado 38
Ilustración 32. Implementación de un módulo en el dispositivo 39
Ilustración 33. Diagrama de bloques del sistema completo 40
Ilustración 34. Zedboard con transmisor y receptor..... 41
Ilustración 35. Interfaz del SDK de Xilinx..... 41
Ilustración 36. ILA integrado en Vivado en ejecución 42
Ilustración 37. IP Cores aislados (7)..... 44

1. Introducción

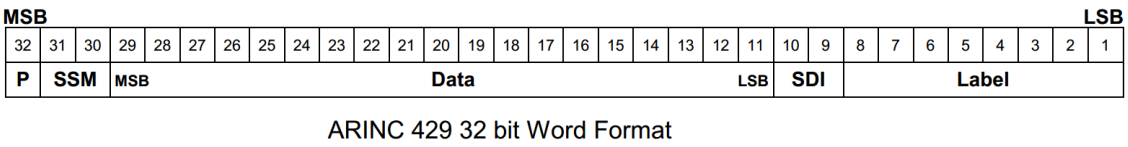
Este proyecto es una pequeña parte de un proyecto mucho mayor que está realizando la empresa Orbital Aerospace y que consiste en diseñar y construir una plataforma hardware pre-certificable en la que poder hacer testeo de aplicaciones de un sistema de aviación o incluso para ir montada en un avión. Al tratarse de hardware para un sistema de aviónica y querer que sea pre-certificable hay que seguir las normas establecidas en 3.3 RTCA/DO 254 (1).

Esta plataforma abarataría mucho los costes de desarrollo de software crítico ya que se podrían probar y correr test directamente sobre esta plataforma sin tener que depender de un banco de pruebas externo. Esta dependencia es una causa muy importante de la demora y coste de desarrollo en este campo ya que los grupos de desarrollo y testeo pierden un porcentaje muy elevado de tiempo del proyecto en conseguir acceso a un banco de pruebas .

En esta memoria se describe el diseño y la implementación de un receptor ARINC-429 además de las múltiples simulaciones y procesos aplicados sobre él. El diseño de este receptor se decidió implementar en una FPGA y programarlo en VHDL descomponiendo el IP Core en varios módulos. Estos módulos deberán pasar una serie de testbenches que garanticen el correcto funcionamiento de cada módulo por separado y del conjunto que forman para asegurarnos de que una vez forme parte de la plataforma hardware, ésta esté preparada para ser certificada.

ARINC 429 es un bus de datos muy común en aviónica. Se trata de un bus simplex sobre un par de cobre con un solo transmisor y hasta 20 receptores cuya codificación es RZ bipolar. Este bus tiene dos velocidades de funcionamiento, a baja velocidad entre 12 - 14.5 Kbps o a alta velocidad alcanzando 100 Kbps. El formato de las palabras enviadas por este protocolo, como se puede ver en la Ilustración 1, es de palabras de 32 bits de los cuales los 8 primeros son de etiqueta, los bits 9 y 10 corresponden al Source/Destination Identifier (SDI), los bits comprendidos entre el 11 y el 29 son los datos, los bits 30 y 31 corresponden al Sign/Status Matrix (SSM) y el bit 32 es el de comprobación de paridad. De todos estos campos solo son obligatorios dos que son el bit de paridad y la etiqueta. El protocolo establece que tiene que haber un espacio de tiempo entre cada palabra enviada al que se denomina GAP.

En la Ilustración 1 se muestra el formado de palabra de 32 bits utilizado en el protocolo ARINC 429.



ARINC 429 32 bit Word Format

Ilustración 1. Formato de palabra en ARINC 429

Este receptor se implementará en una FPGA (Field Programmable Gate Array). Una FPGA es un circuito integrado que contiene bloques de lógica configurable lo que la convierte en una herramienta muy potente que nos permite diseñar e implementar hardware complejo

de una manera relativamente sencilla. Este diseño se realiza gracias a un lenguaje de descripción hardware o HDL por sus siglas en ingles. Estos lenguajes permiten un alto nivel de abstracción a la hora de realizar un diseño hardware ya que son similares a los lenguajes de programación pero con la diferencia de que gracias al programa sintetizador podemos convertir esa programación en su circuito equivalente. En este caso vamos a emplear VHDL por su robustez.

Ya existen en el mercado COTS de IP Cores receptores A429 pero su precio es muy elevado (15000$ de LogiCircuit y 16000€ de DEVELeast) y a pesar de tener que invertir mucho tiempo en su realización, podemos ver que merece la pena embarcarse en este proyecto. No solo por el importante ahorro económico si no que pese al gasto de tiempo, 4 meses de duración estimada, podrían compensarse con creces con una sola venta de este IP Core al margen de las posibles ganancias del proyecto global.

2. Objetivos

Los objetivos marcados para este proyecto son implementar cinco módulos VHDL junto con otros módulos del catálogo de Xilinx que permitan establecer una conexión entre una aplicación corriendo en el procesador y otro modulo ARINC 429 transmisor. La conexión entre el procesador y el IP Core receptor ARINC 429 se realizará a través de un bus AXI la cual también deberá ser implementada. Se empleará la herramienta proporcionada por Xilinx Vivado para el diseño del IP Core y para las simulaciones se usarán tanto Modelsim como Xsim, esta última integrada en el propio Vivado.

El IP Core receptor ARINC 429 estará compuesto por un módulo AXI esclavo, un módulo que contenga la lógica de control al que llamaremos Logic Control, una FIFO de 64 bits y 64 palabras, un módulo que sea un receptor de línea o lo que es lo mismo un receptor que detecta los bits y que además se encargue de calcular la paridad de cada palabra y generar los diferentes tipo de errores si los hubiese, otro módulo que será encargado de darle la vuelta a la etiqueta, otro módulo cuya funcionalidad es la de comprobar si la etiqueta de la palabra entrante está habilitada y por tanto, la palabra debe ser enviada al registro y un último módulo que se encargue de hacer de intermediario entre la FIFO y el bus AXI.

Una vez realizados estos módulos de los cuales el "Control Logic", la FIFO, el Receptor de Línea, el "Latch" o volteador de etiqueta, el Comprobador de Etiquetas y el modulo "Fifo To Output Register" estarán englobados en un mismo módulo al que llamaremos módulo receptor. El esclavo AXI estará interconectado con este modulo receptor y ambos a su vez formarán un módulo llamado AXI_A429_RX_v1_0.

En la Ilustración 2 se muestra el diseño preliminar del receptor a implementar con sus módulos y su posible disposición. Al Receptor de Línea llegaran los bits provenientes del bus y el Registro de la Fifo a la Salida (fifo to output register) será el encargado de enviar las palabras de 32 bits al esclavo AXI.

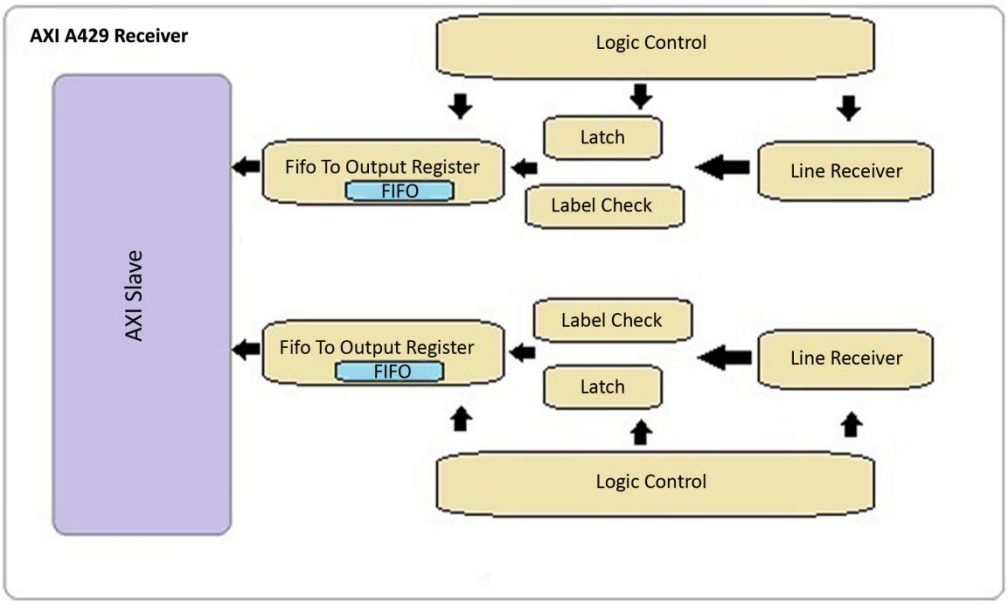


Ilustración 2. Diseño preliminar del receptor A429 con los módulos a implementar

De forma paralela a la implementación de estos módulos en VHDL se irán implementando y ejecutando unas simulaciones (testbenches) básicas que comprueben las funcionalidades de cada módulo a medida que se generen.

Una vez implementados los módulos y comprobadas sus funcionalidades básicas se procederá a la implementación de testbenches automáticos que comprueben la funcionalidad completa de cada módulo así como el conjunto de todos los módulos llamado módulo Receptor.

Tras haber realizado satisfactoriamente las simulaciones se programará en la FPGA y será depurada (debug) mediante una herramienta incluida en Vivado. Se probará su funcionamiento y se mejorará el diseño hasta que la depuración funcione correctamente.

El objetivo último de este proyecto es enlazarlo con otro que consiste en la realización de un transmisor ARINC 429, es decir, implementar ambos proyectos en la FPGA y conseguir mediante una aplicación corriendo en el procesador una comunicación entre el módulo transmisor y el receptor.

3. Metodología

3.1 Implementación de Módulos y Testbenchs

Para la implementación del IP Core se generará código VHDL de los módulos diseñados empleando una metodología de diseño de dos procesos. Este método permite reducir tiempo de diseño y evita los típicos problemas inherentes a los sistemas diseñados con el método comúnmente empleado más parecido a la programación estructurada. Lo que se pretende conseguir con el uso de este método es un mecanismo uniforme de codificación de algoritmos, aumentar el nivel de abstracción, facilitar la comprensión del código, simplificar su depuración y conseguir ver de una manera más clara las conexiones entre componentes en el análisis RTL. Es muy sencillo entender el funcionamiento de este método, solo hay que cumplir dos condiciones y una estructura. Las condiciones son emplear tipos record (record types) en todos los puertos y declaraciones de señal y solo utilizar dos procesos por módulo. Y la estructura a seguir se muestra en la Ilustración 3 que se encuentra al final de este apartado.

Las simulaciones se ejecutarán en modo comportamiento (behavioral) y una vez funcionen al 100% se pasará a ejecutar en modo funcional (functional) y temporal (timming). El método que se empleará para evaluar estas simulaciones será mediante archivos. Estos serán generados mediante un script matlab y contendrán los estímulos de entrada para cada módulo. El testbench se encargará de escribir un archivo de salida con las respuestas provocadas por cada módulo y se terminarán automáticamente al probar todos los estímulos posibles. Por medio de otro script matlab se comprobará que la respuesta del módulo a cada uno de los estímulos de entrada ha sido la que se esperaba y en base a esto el script decidirá si el módulo funciona o no correctamente.

Para introducir estos estímulos en cada módulo los testbench tendrán que ser capaces de leer los archivos de entrada y de escribir un nuevo archivo de salida. Esto lo haremos en todos los testbench siguiendo el modelo de la Ilustración 4.

En la Ilustración 3 se ve la estructura de un módulo diseñado con el método de dos procesos. El primer proceso solo contiene en su lista de sensibilidad el reloj y dentro simplemente en el flanco de subida actualiza las variables. En el segundo proceso, como vemos, su lista de sensibilidad contiene la variable r de tipo record, el reset y las entradas del módulo. Entrando en el proceso vemos que dependiendo de en qué estado se encuentre el módulo podremos programar un comportamiento u otro. En este caso simplemente del estado de espera pasara al estado de gap y viceversa. Al final de este proceso si el reset esta activado se inicializa todo y por último se actualizan las variables y las salidas. Hay que tener en cuenta que el hecho de actualizar las variables debe hacerse con valores registrados y no con otras variables ya que en este último caso se puede llegar a producir problemas, incurriendo en un comportamiento no deseado de nuestro módulo.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Entidad is
  Port (
    clk          :  in  STD_LOGIC;
    rst          :  in  STD_LOGIC;
    input1       :  in  STD_LOGIC;
    output       :  out STD_LOGIC
  );
end Entidad;

architecture Behavioral of Entidad is
  type STATE_TYPE is (idle,transmitting, gap);
  -- USO DE TIPOS RECORD

  type reg_type is record
   state          :  STATE_TYPE;
   entrada1       :  STD_LOGIC;
   salida1        :  STD_LOGIC;
   unaVariable    :  Integer;
  end record;

  signal r, rin : reg_type;

begin
  -- PRIMER PROCESO

  process(clk)
   begin
      if rising_edge(clk) then
         r <= rin;
```

```
-- SEGUNDO PROCESO

process(r,rst,input1)
   variable v : reg_type;
 begin
   v := r;
   --inicializar variables

   case r.state is
      when idle => v.state:=gap;

      when transmitting =>

      when gap => v.state:=idle;

      when others => null;
   end case;

   -- reset activado
   if (rst = '0') then
      v.state:=idle;
      v.salida1:='0';
   else
   end if;

   --registrar variable
   rin <= v;

   --salidas
   salida1 <= r.salida1;
```

Ilustración 3. Estructura tipo del método de diseño de dos procesos

En la Ilustración 4 podemos ver la estructura seguida en todos los testbenches automáticos realizados en este proyecto. Al comienzo del testbench hay que añadir las librerías necesarias para la escritura que son IEEE.STD_LOGIC_TEXTIO, y std.textio y en el caso de querer escribir alguna variable de tipo Integer antes deberá ser pasada a STD_LOGIC_VECTOR para lo que necesitaremos la librería IEEE.NUMERIC_STD. El siguiente paso es implementar los códigos que desempeñan las funciones de reloj y reset. Por último leemos los datos escritos en un archivo línea a línea y almacenamos los caracteres, predeciblemente ceros y unos, en la variable de entrada de nuestro componente. Mantenemos ese valor en la variable durante, en este ejemplo, 20 nanosegundos y la cambiamos a la siguiente cadena escrita en nuestro archivo. Y este proceso se repite hasta el número de líneas especificado. Suponemos que nuestro componente reacciona a los estímulos y lo que hacemos es escribir la variable de salida en un segundo archivo. Esto ocurrirá hasta que cambie la salida y haya pasado 1 milisegundo ya que entonces el testbench parará al saltar la alerta de fallo grave.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_TEXTIO.ALL;
use std.textio.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Entidad_autoTestbench is
end Entidad_autoTestbench;

architecture Behavioral of Entidad_autoTestbench is
   component Componente is
      Port (
            ACLK      : in std_logic;
            ARESETN : in std_logic;
            input1     : in STD_LOGIC_VECTOR(31 downto 0);
            output1    : out std_logic
      );
   end component;

   signal clk                    : STD_LOGIC;
   signal resetn                 : STD_LOGIC;
   signal input1_signal          : STD_LOGIC_VECTOR(31 downto 0);
   signal output1_signal         : STD_LOGIC;
   signal I_want_to_finish_autoTestbench :STD_LOGIC:='0';
begin

Componente_inst : Componente
   PORT MAP (
      ACLK          => clk,
      ARESETN       => resetn,
      input1        => input1_signal,
      output1       => output1_signal
   );

generate_reset : process
   begin
      resetn <= '0';
      wait for 100 ns;
      resetn <= '1';
      wait for 1ms;
      I_want_to_finish_autoTestbench<='1';
      wait;
```

```
generate_clock : process
   begin
      clk <= '1';
      wait for 1 ns;
      clk <= '0';
      wait for 1 ns;
   end process;
writing_in_file : process
      function value_to_bin(c : character) return std_logic is
      begin
         if(c='1') then return ('1');
         else return ('0');
         end if;
      end;
      file DataInput: TEXT open read_mode is "Path\documento.txt";
      variable DataLine : line;
      variable v : std_logic;
      variable n_columnas_archivo:Integer:=32;
      variable n_filas_archivo:Integer:=32;
   begin
    for i in 1 to n_filas_archivo loop
      readline (DataInput,DataLine);
      for j in 1 to n_columnas_archivo loop
         v:= value_to_bin(DataLine(n_columnas_archivo+1-j));
         input1_signal(j-1) <= v;
      end loop;
      wait for 20 ns;
    end loop;
    wait;
   end process;

reading_from_file : process
   file OutputFile : TEXT open write_mode is "Path\documento2.txt'
   variable outLine : line;
   begin
      wait until output1'EVENT;
      write(outLine,output1);
      writeline(OutputFile,outLine);
      if I_want_to_finish_autoTestbench='1' then
         file_close(OutputFile);
         assert false report "end of simulation" severity failure;
      else
      end if;
   end process;
```

Ilustración 4. Estructura tipo del método empleado para realizar testbenches automáticos

3.2 Agilismo Aplicado al Proyecto

Para la realización del proyecto también se ha puesto en práctica una metodología de desarrollo ágil (2) derivada de la denominada Scrum (3). Se trata de un conjunto de buenas prácticas diseñadas para agilizar y optimizar el trabajo. En Scrum se definen unos roles que son el gestor principal (Scrum Master), cliente (Product Owner) y el equipo (Team). También se dividen las tareas en sprints que duran entre 3 y 4 semanas durante los cuales el equipo no puede ser molestado con actualizaciones de requisitos sobre el producto. Se realizan reuniones todos los días de 15 minutos para que cada miembro del equipo sepa lo que ha hecho cada uno de sus compañeros y lo que tiene pensado hacer ese mismo día. Por esto último se realizan al comienzo de la jornada. A parte de esta reunión diaria, se realiza una reunión de mayor duración cada semana para hablar de los problemas y aciertos de la semana y otra reunión al comienzo de cada sprint para planear y definir las tareas del siguiente.

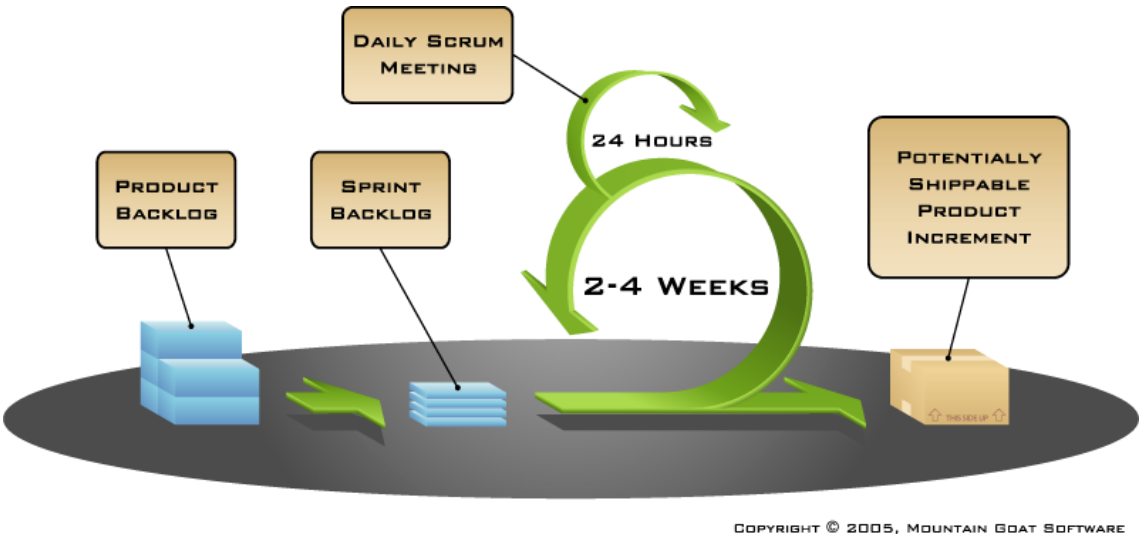


Ilustración 5. Definición gráfica de la distribución del tiempo en un proyecto llevado con Scrum (4)

La aplicación de este método en este proyecto ha sido realizada siguiendo un patrón parecido. Al tratarse de un proyecto de I+D interno de la empresa, el rol de cliente fue asumido por el director de I+D. El rol de gestor principal fue adjudicado al jefe del proyecto al contar con una visión global del proyecto. Como en todo proyecto llevado mediante Scrum, al comienzo se hizo un listado con las tareas a realizar a lo largo de 4 meses y se estableció una duración aproximada a cada tarea. Se realizó un Gant y se fijaron 4 fases.

Una primera fase de familiarización con las herramientas de uso y aprendizaje de protocolos de aviónica y lenguaje de descripción hardware elegido (VHDL). Además esta primera fase incluía el aprendizaje de procesos de calidad realizados por la empresa así como de metodologías de implementación y trabajo.

La segunda fase consistió en el diseño e implementación de los módulos que componen el receptor A429 empezando por el receptor de línea pasando por el "Latch" o inversor de etiqueta, el comprobador de etiquetas y el fifo al registro de salida y terminando

por el bloque con la lógica de control. A esto hay que añadir los testbenches de comprobación básica.

A continuación la tercera fase y la más larga, engloba la realización de los testbenches automáticos así como los scripts matlab de comprobación de resultados. Esta fase lleva implícita la corrección de errores en módulos además de la ejecución de todos los métodos de simulación necesarios para evaluar el funcionamiento del sistema completo.

La cuarta y última fase consiste en la programación del IP core en una FPGA y probar su funcionamiento contra un transmisor A429 implementado en la misma. También se comprobarán el flujo de datos en tiempo real en el receptor implementado. En esta última fase adquirirá un papel protagonista la redacción de la documentación necesaria.

3.3 RTCA/DO 254

La norma RTCA/DO254 es una metodología para desarrollar hardware electrónico cuando el fallo de dicho sistema puede provocar fallos catastróficos en la integridad del avión o daños mortales en las personas.

Este tipo de normas surgen debido a la incapacidad de calcular matemáticamente la probabilidad de fallo de un sistema tan complejo. De este modo, y dado que los sistemas embarcados en aviones deben tener una probabilidad de fallo menor a una entre un billón, es necesario seguir una estricta normativa que garantice que al seguirla, la probabilidad de fallo del sistema sea inferior a la cifra anteriormente mencionada.

Dentro de la normativa RTCA/DO 254 se puede discernir entre 5 niveles de criticidad denominados DAL y que van desde el A hasta el E. El nivel A es el nivel más estricto, es decir el que cuyo fallo del sistema provoca un daño catastrófico en el avión mientras que el E, es el nivel en el que el fallo no afecta a la seguridad del avión.

Para garantizar que la probabilidad de error sea menor a una entre un billón, se especifica en la norma que se deben seguir 5 pasos. El primer paso es la captura de requisitos, es decir, especificar en primer lugar todo lo que queramos que sea capaz de hacer el sistema a desarrollar. El segundo paso es la realización del diseño conceptual, es decir, se explica un poco por encima, como se va a desarrollar el sistema para que realice las funciones especificadas en los requisitos. El tercer paso es el diseño detallado, donde ya se explica cómo se va a realizar todo en profundidad. En el caso de FPGA, sería especificar cómo se van a realizar los diferentes módulos, conexiones, etc. El cuarto paso es el diseño e implementación de lo descrito anteriormente. El último paso, es la verificación de que el sistema ya diseñado, realiza correctamente las funciones que han sido definidas en los requerimientos. Si por cualquier circunstancia en alguno de estos pasos se produjese un fallo, bien sea por culpa del mal desarrollo del sistema o porque el requisito estaba mal definido, se deberá volver hasta el punto de fallo y volver a realizar todos los pasos, de ese en adelante. A estos pasos se les conoce coloquialmente como "la V".

4. Desarrollo e Implementación

4.1 Introducción

A la hora de comenzar este proyecto se decidió por su robustez usar el lenguaje de descripción hardware VHDL frente a la alternativa Verilog. Al tener poca experiencia con esta clase de lenguajes, la primera tarea de implementación fue la de familiarizarse y conocer más a fondo las herramientas de programación de Xilinx (Vivado) y aprender las reglas y metodologías básicas de implementación como las ya descritas en el apartado 3.1 Implementación de Módulos y Testbenchs, debido a que dentro de toda la metodología de VHDL, muy pocas construcciones son sintetizables. A esta primera tarea hay que añadirle la de documentación sobre el protocolo ARINC 429 y las características técnicas necesarias de un transmisor y receptor en este protocolo.

4.2 Definición Técnica de un Receptor ARINC 429

El receptor ARINC 429 contará con dos líneas de recepción que se ajusten a la especificación y podrán ser usadas como caminos redundados o independientes. Cada línea de recepción tendrá dos posibles velocidades de recepción:

- Alta velocidad → 100 Kbps
- Baja velocidad → 12.5 Kbps

El interfaz con el procesador será un esclavo AXI, y los interfaces externos del receptor podrán conectarse directamente a un controlador de línea comercial como por ejemplo HOLTIC HI-844x, HI-845x o Intersil HS3182. En los Objetivos ya hemos podido ver un diseño preliminar al que le faltaba contenido al esclavo AXI. En cada receptor habrá un módulo FIFO con capacidad para 64 palabras de 64 bits.

Para implementar este receptor completo formado por dos receptores individuales, se implementará en cada receptor individual una línea con un esclavo AXI y una vez terminado se duplicará el diseño y se englobarán ambos en un único IP Core.

En la Ilustración 6 vemos que el módulo esclavo AXI contiene once registros que están directamente relacionados con el resto de módulos del receptor. El módulo que contiene la lógica de control necesita conocer la información escrita en el registro de control del AXI para poder actuar en consecuencia y a su vez el comprobador de etiquetas debe conocer el mapeo de etiquetas de dichos registros del esclavo AXI. Además el registro de estado deberá reflejar el estado actual del receptor (estado del receptor, errores de recepción, estado de la FIFO). En este diseño de definición técnica no entramos en mayor profundidad a la hora de especificar la interconexión entre módulos.

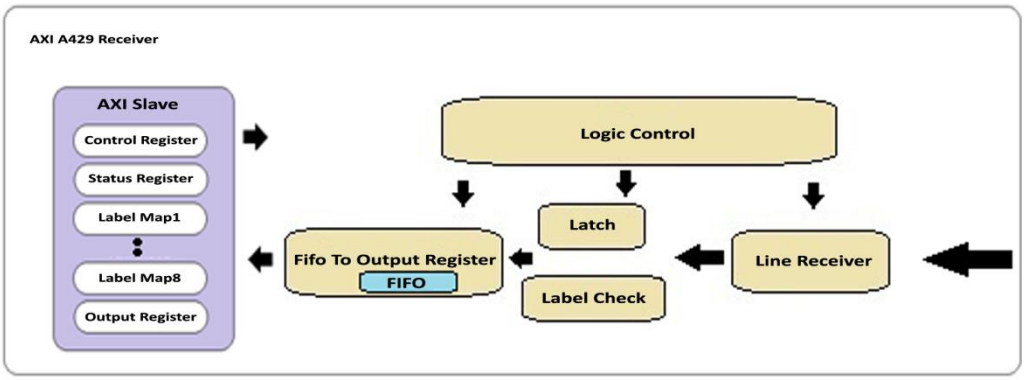


Ilustración 6. Diseño preliminar con interconexiones entre registros del AXI y los módulos del receptor

Para la FIFO usaremos un IP Core del catalogo de Xilinx llamado FIFO Generator. Con este módulo podemos generar la FIFO que necesitamos fácilmente a través de una GUI como vemos en la Ilustración 7.

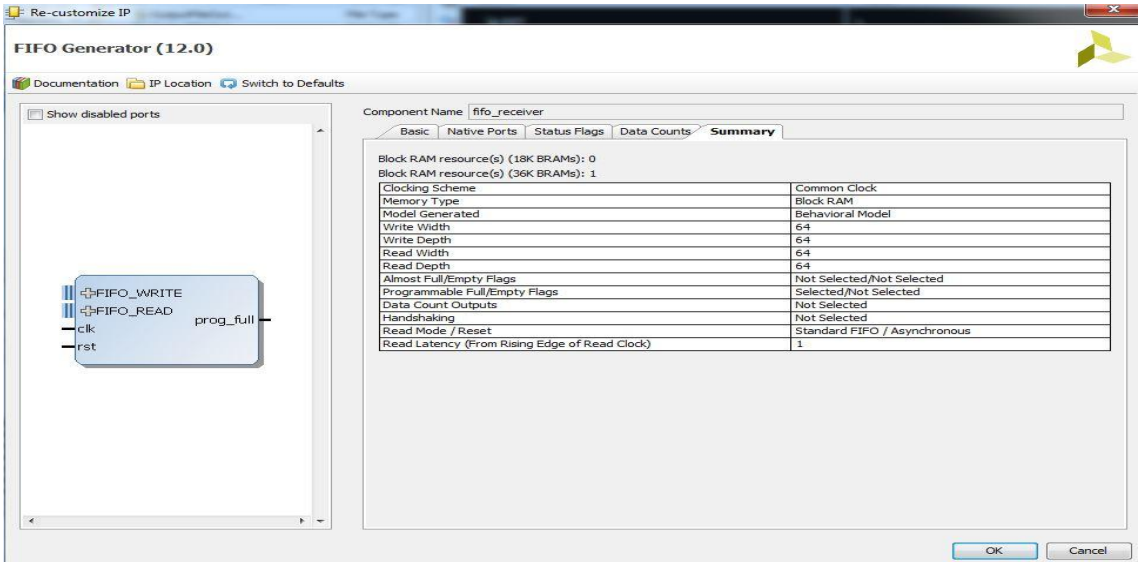


Ilustración 7. GUI de configuración del módulo FIFO Generator

En dicha ilustración, la pestaña seleccionada es la de resumen y vemos que las especificaciones de nuestra FIFO coinciden con las especificadas anteriormente. Capacidad de albergar 64 palabras de 64 bits.

Las especificación del protocolo ARINC 429 define que las señales de salida estarán codificadas con el código RZ Bipolar con unos parámetros de tolerancia indicados en la Tabla 1 e Ilustración 8.

La Ilustración 8 puede servirnos de apoyo gráfico para entender los rangos de tolerancia admitidos según la especificación ARINC 429. Como vemos el tiempo de flanco de subida se considera a partir del 10% del valor de amplitud y hasta el 90% del mismo al igual que pasa con el tiempo de flanco bajada.

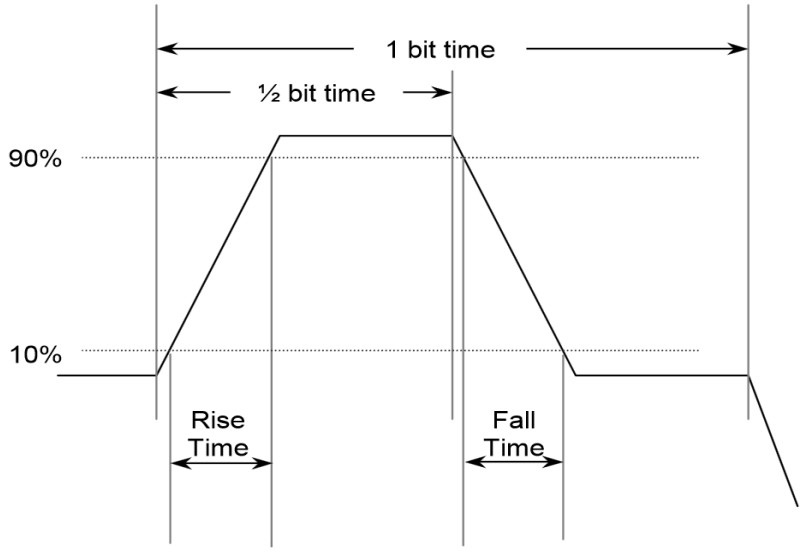


Ilustración 8. Gráfico de pulso para definición de las tolerancias de la línea de salida (5)

En la Tabla 1 podemos ver los datos concretos de tolerancia especificados por la norma. En este proyecto hemos elegido una frecuencia de operación a baja velocidad de 12.5 Kbps y por tanto el periodo de bit será de 80 µseg con una tolerancia de 2 µseg.

Parámetro	Operando a alta velocidad	Operando a baja velocidad
Tasa de bits	100 Kbps ± 1%	12 - 14.5 Kbps
Tiempo 1 bit	10 µseg ± 2.5%	(1/Tasa de bit) µseg ± 2.5%
Tiempo ½ bit	5 µseg ± 5%	(1 bit time/2) ± 5%
Tiempo de flanco de subida	1.5 µseg ± 0.5 µseg	10 µseg ± 5 µseg
Tiempo de flanco de bajada	1.5 µseg ± 0.5 µseg	10 µseg ± 5 µseg

Tabla 1. Tolerancias definidas en la especificación ARINC 429 (5)

Entre recepción de palabras, el receptor tiene un tiempo de espera entre dos palabras en el que ambas líneas están a nivel lógico bajo. A este tiempo de espera le llamamos gap y por defecto será de cuatro veces el periodo de bit, aunque puede variar de 0 a 7 periodos de bit si así el software lo requiera. De esta manera nos aseguramos el no necesitar el envío de una señal de reloj adicional para sincronizar transmisor y receptor

4.3 Definición concreta del interfaz esclavo AXI

El interfaz esclavo AXI es el módulo que permite la comunicación entre el receptor ARINC 429 y el software ejecutado en el procesador al otro lado del AXI. Este módulo contiene once registros y como vimos en la Ilustración 6 son un registro de control, un registro de estado, ocho registros de mapeo de etiquetas y un registro de datos.

Vivado nos permite generar un esclavo AXI personalizado así que nos valdremos de la GUI para crear el modelo básico y ampliar lo que sea necesario.

En la Ilustración 9 vemos el primer paso para crear un periférico AXI. Si seguimos la GUI podremos especificar el tamaño, la profundidad y la cantidad de registros que queremos incluir en nuestro periférico AXI. Además nos da la posibilidad de establecer nuestro módulo como maestro o esclavo. Para este proyecto necesitamos un periférico AXI esclavo de once registros de 32 bits y de una sola palabra de profundidad.

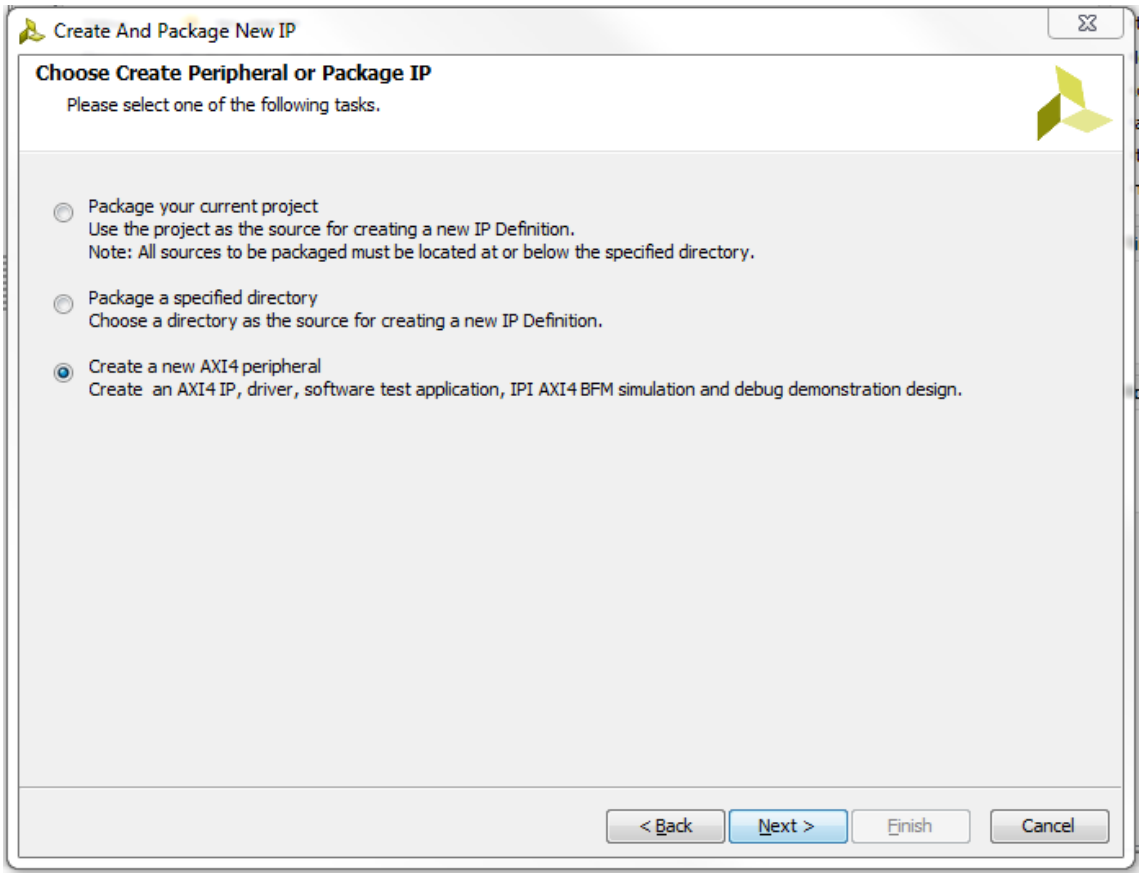


Ilustración 9. GUI de generación de esclavo AXI básico

En la Tabla 2 tenemos descrita la funcionalidad de cada bit del registro de control. El bit 0 activa o desactiva el receptor, es decir actúa como si estuviera reseteado. Con el bit 0 a 1 lo que conseguimos es que los bits se detecten y se calculen los posible errores. Los bits de paridad determinan si comprobar la paridad y el tipo de paridad a recibir al igual que el bit de tasa de bit determina la frecuencia de recepción. Los gap se controlan con los bits del 4 al 7. Si el bit 4 está a 1, la duración del gap vendrá determinada por el número binario formado por

los bits del 5 al 7 siendo el 5 el LSB, obteniendo un rango del 0 a 7 periodos de bit posibles para la duración del gap.

Bit	Función	Estado	Descripción
CR0	Estado del Receptor	0	Desactivar
		1	Activar
CR1	Memoria de Etiquetas	0	No configurado, recibir todas las etiquetas
		1	Filtrar las etiquetas que se especifican en los registros de mapeo de etiquetas
CR2	Activar comprobación de paridad	0	Desactivar
		1	Activar
CR3	Tipo de Paridad	0	Paridad impar
		1	Paridad par
CR4	Controlador de Gap	0	4 veces el periodo de bit (Por defecto)
		1	Duración especificada en bits CR5 - CR7
CR5 - CR7	Periodo de Gap	0 - 7	GAP = Valor * Periodo de bit
CR8	Tasa de Bit	0	Baja velocidad 12.5 Kbps
		1	Alta velocidad 100 Kbps
CR9:CR16	Tolerancia de Subida	0-0xFF	Numero de ciclos antes de muestrear valores
CR17:CR24	Periodo de Tolerancia	0-0xFF	Ciclos del periodo de tolerancia
CR25	Resetear la FIFO	0	Sin acción
		1	Resetear

Tabla 2. Bits y funcionalidad desempeñada por el registro de control del esclavo AXI

En la Tabla 3 podemos ver como este registro refleja en cada momento el estado del receptor. Si esta activado, si hay errores , el estado de la FIFO en cada momento o si ha leído el dato el procesador .

Bit	Función	Estado	Descripción
ST0	Estado del Receptor	0	Desactivado
		1	Activado
ST1	Error de Paridad	0	No hay error
		1	Error de paridad del mensaje
ST2	Error de Gap	0	No hay error
		1	Gap menor del especificado
ST3	Error de Frecuencia	0	No hay error
		1	Error frecuencia de reloj
ST4	Error sumatorio Bits	0	No hay error
		1	Numero erróneo de bits recibidos
ST5	FIFO medio Llena	0	No medio llena
		1	Medio llena
ST6	FIFO Llena	0	No llena
		1	Llena
ST7	FIFO Vacía	0	No vacía
		1	Vacía
ST8	Dato Preparado	0	Registro de salida vacío
		1	Registro de salida contiene un dato válido. Poner a cero cuando se lee el registro de salida

Tabla 3. Bits y funcionalidad desempeñada por el registro de estado del esclavo AXI

Por último contaremos con un registro de datos o salida en el cuál se almacenarán 32 bits provenientes del receptor y que se encargará de leer el procesador a través del software. Los 31 primeros bits serán datos y el bit 32 será el mismo bit que el bit ST1 del registro de estado. Estos datos serán una pequeña modificación de los datos del formato establecido en ARINC-429 que ya vimos en la Ilustración 1 y que quedará como se muestra en la Ilustración 13.

4.4 Uso del IP Core

Tras desactivar el reset poniéndolo a 1, el receptor A429 estará listo para recibir la configuración, los mapas de etiquetas y recibir datos.

El primer paso es activar el receptor a través del bit del registro de control correspondiente (bit CR0, Tabla 2) y obtener los datos del resto de parámetros permitidos por dicho registro como son el control de paridad, la tasa de bit, el periodo de gap, los periodos de subida y tolerancia y el reseteo de la FIFO.

En la Ilustración 10 se puede apreciar el esquemático del receptor completo con sus puertos de salida y de entrada.

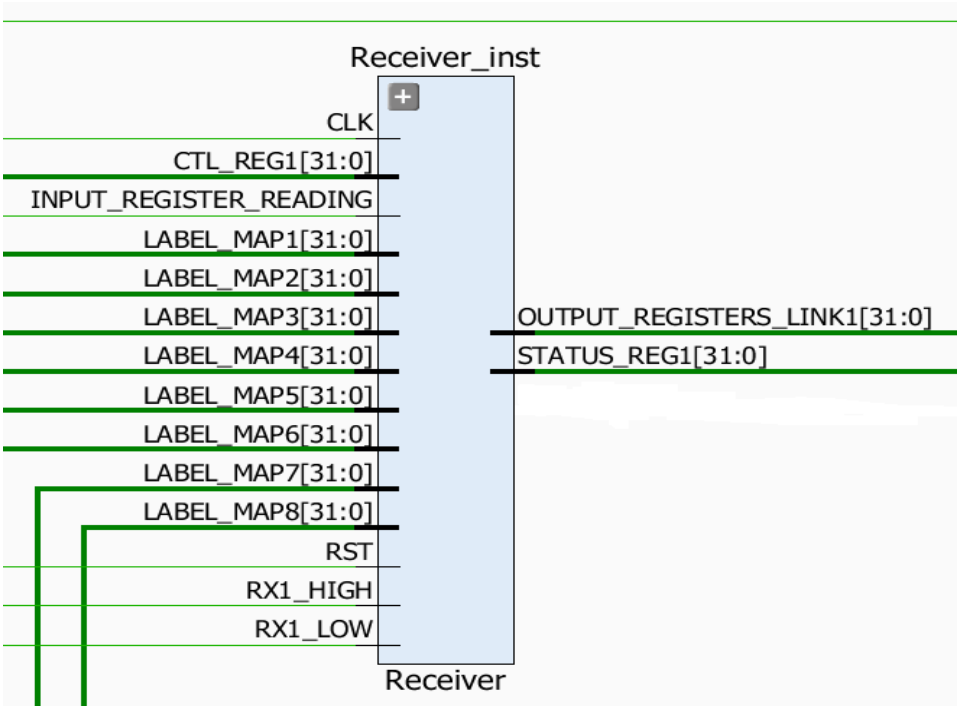


Ilustración 10. Receptor ARINC 429 completo

Mientras el bit CR0 este activo el receptor podrá recibir datos y transferirlos si procede al registro de datos del AXI. El receptor irá recibiendo bits y procesándolos hasta que llegue el bit 32 de cada palabra. Si el CR1 está puesto a 0 para que reciba todas las etiquetas o bien está a 1 y la etiqueta se encuentra dentro del mapa de etiquetas, la palabra y los posibles errores que contenga dicha palabra se envían a una FIFO en la que quedarán almacenados (salvo que se resetee la FIFO) a la espera de que el procesador mande una señal de que ya ha leído los datos que tiene almacenados en el registro de datos del AXI. Una vez se ha terminado de recibir la palabra de 32 bits se comprueba el número de gaps para ver si se está produciendo un error mientras se espera a recibir la siguiente palabra. Cuando el procesador lee los datos de su registro de salida, se envía una señal a la FIFO para que salga otro dato almacenado y se envíe al registro de datos del AXI.

4.5 Funcionamiento de los Módulos del Receptor

4.5.1 Line Receiver

El módulo Line Receiver es el primer módulo en el que se adentran los bits que llegan del bus y por tanto; es el módulo encargado de discernir entre 1's 0's y nulls (ceros en ambas líneas). Además este módulo es el que implementa la lógica más compleja, puesto que a su vez debe generar la paridad de la palabra y comprobar si hay algún tipo de error ya sea de gap, de frecuencia o de número erróneo de bits.

En este módulo entran los bits CR0, CR4 o CR5-7,CR8, CR9-16 y CR17-24 (Tabla 2) del registro de control además de la señal de reloj, del reset y de líneas alta y baja del bus. Como puertos de salida de este módulo se encuentran la salida de la palabra de 32 bits la paridad de la palabra, los posibles errores de gap, frecuencia y número de bits y la salida que indica a los demás módulos cuando son válidos los datos de los puertos antes mencionados. Todo esto se puede apreciar en la Ilustración 11 .

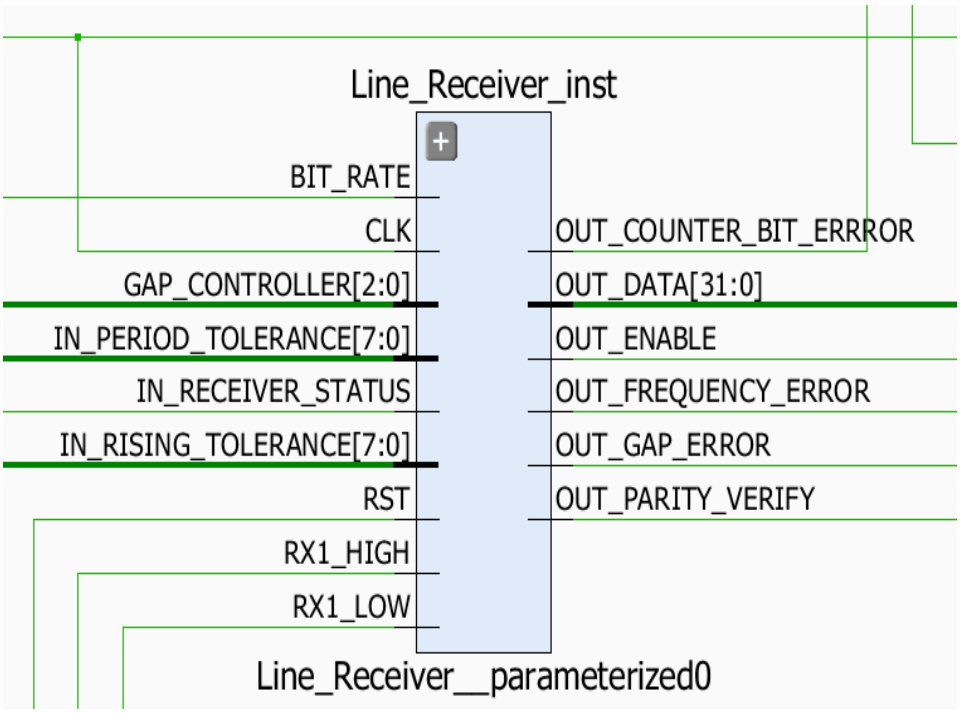


Ilustración 11. Esquemático del módulo Line Receiver

El Line Receiver funciona internamente de la siguiente manera; cuando se detecta que llega un bit por las líneas, bien sea porque se ha puesto a 1 la línea alta o la baja, se espera el número de ciclos de reloj especificado por la tolerancia de subida, se muestrea para conocer si el bit es un 0 o un 1, se envía a un registro de de desplazamiento de izquierda a derecha donde se van almacenando los bits y se hace la "*xor*" de ese bit recibido con los anteriores tras lo cual, se almacena en una variable. Este registro de desplazamiento tiene este sentido y no desplaza de derecha a izquierda ya que la palabra se debe almacenar tal como se indica en la Ilustración 1 (a excepción de que el bit 32 debe ser el ST1 y no el bit de paridad) y en cambio

se recibe la palabra como se muestra en la Ilustración 12. Tras esto, se espera a que pasen ciclos de reloj, hasta llegar al final del periodo de bit menos los ciclos marcados en el periodo de tolerancia y se comprueba que para ese tiempo no haya llegado ningún bit, si por contra ha llegado, se vuelve a repetir el proceso explicado anteriormente y se marca que ha habido un error de frecuencia. Si por contra, no ha habido ningún error de frecuencia, se espera a que llegue un nuevo bit hasta que termine el periodo de bit y cuando éste llegue, se suma un bit al contador de bits y se repite el ciclo. Una vez que se han recibido los 32 bits y se va pasar al periodo de gap entre palabras, se establece a 1 la línea OUT_ENABLE para que los módulos posteriores sepan que es el momento de adquirir los datos del Line Receiver.

Si tras pasar al periodo de gap se recibe un bit, debido a que ya han habían llegado los 32 de la palabra, se interpreta como que llega el bit 33 y se pone el error de contador de bits a 1. Si por contra esto no ocurre y se reciben nulls, se espera a que vayan pasando hasta el número especificado en el periodo de gap. Si se recibe un bit antes de que el número de periodos de gaps transcurridos sea igual al número especificado en el registro de control se activa el error de gap y si no llega antes, se mantiene a cero y se pasa a la espera de recibir una nueva palabra.

Por motivo de robustez del sistema se ha decidido que solo se pueda cambiar de velocidad de recepción y el número de periodos de gap cuando el receptor se encuentra desactivado.

A continuación se muestran tanto el formato en el que llegan al receptor cada palabra y el formato en el que se almacena en el registro de salida o datos.

Como ya se ha mencionado anteriormente, en la Ilustración 12 se puede ver en la manera en la que llegan los bits al receptor, tomando como primer bit de llegada el octavo bit o MSB de la etiqueta. La Ilustración 13 muestra la manera en la que la palabra de 32 bits se almacena en el registro de salida y como el bit 32, que en un principio contenía la paridad de la palabra, ha sido sustituido por el bit ST1.

8	7	6	5	4	3	2	1	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Label								SDI		LSB	Data																	MSB	SSM		P

Ilustración 12. Formato de recepción de palabra en A429 en sentido de izquierda a derecha.

MSB																															LSB
32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
ST1	SSM		MSB									Data									LSB	SDI						Label			

Ilustración 13. Formato en A429 de almacenamiento de palabra en registro de salida.

4.5.2 Latch

El módulo Latch es el encargado de coger la palabra de 32 bits que llega del Line Receiver y darle la vuelta a los bits de la etiqueta, para ponerlo en orden y así poder almacenar en el orden correcto la palabra tal como se indica en la Ilustración 13.

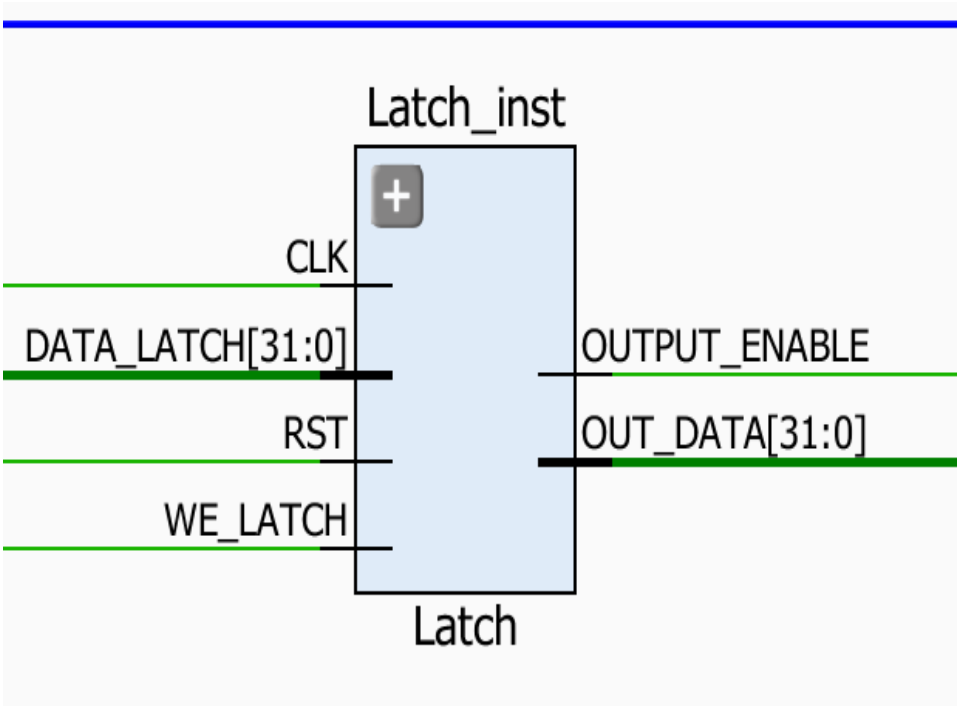


Ilustración 14. Esquemático de módulo Latch

Como puertos de entrada dicho módulo tiene el reloj, el reset, la entrada de datos que viene del Line Receiver y la señal que indica al módulo cuando adquirir los datos que vienen también del Line Receiver. Para los puertos de salida se utiliza la misma estructura que para el módulo anterior, se tiene la salida de la palabra de 32 bits y por otro lado una salida que habilita al siguiente módulo a capturar datos cuando dicha señal se ponga a 1. Ambos puertos de salida se conectan con puertos entrada del módulo Fifo_To_Output_Register.

El comportamiento interno es bastante simple, su única función es la de esperar a que se ponga a 1 la señal de entrada que habilita la captura y cuando esto sucede le da la vuelta a la etiqueta y almacena toda la palabra en un registro interno para ser enviada síncronamente con el siguiente flanco de subida del reloj. Asimismo, cuando eso sucede se pone también a 1 el bit del puerto OUTPUT_ENABLE.

4.5.3 Label Check

La función que desempeña este módulo es la de comprobar si la etiqueta que se encuentra en la palabra de 32 bits está dentro de los mapas de etiquetas y por tanto; si es enviada al registro de salida o no dicha palabra.

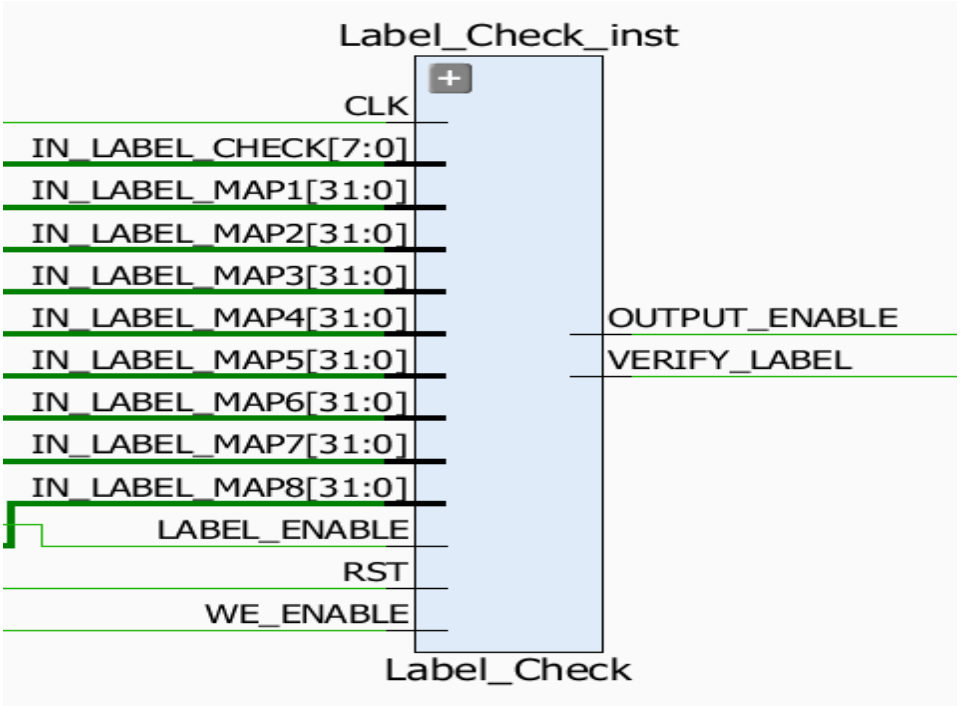


Ilustración 15. Esquemático del módulo Label Check

Aquí entran los 8 mapas de etiquetas que vienen de registros del esclavo AXI, la etiqueta de la palabra de 32 bits, el bit CR1 de la Tabla 2, el reloj, el reset y la señal que habilita a dicho módulo para la adquisición de la etiqueta. Como puertos de salida el sistema se envía al módulo Fifo_To_Output_Enable la verificación de la etiqueta y la señal para indicar al módulo cuando debe adquirir el dato.

El funcionamiento interno de este módulo también es bastante sencillo. Como ya se ha mencionado anteriormente llegan los 8 mapas de etiquetas procedentes del esclavo AXI. Como hay 256 diferentes valores de etiquetas que pueden llegar en una palabra de 32 bits (8 bits de etiqueta), la forma en que se comprueba si está incluida en los mapas de etiquetas es la siguiente: cada bit de cada mapa se corresponde con una etiqueta y por tanto cada mapa contiene 32 etiquetas siendo el mapa 1 el que contiene las etiquetas de la 0 a la 31, el mapa 2 de la 32 a la 63 y así sucesivamente. Se puede ver más claramente en la Ilustración 16.

Ilustración 16. Mapeo de etiquetas

Todos estos mapas de etiquetas se meten en una variable interna y si el bit CR1 está activo, la etiqueta (8 bits) de la palabra de 32 bits se utiliza para acceder a la posición de valor almacenado de todos los mapas de etiquetas. Si este valor se encuentra a 1 se saca un 1 a la salida y si está a 0 se saca un 0. Si por contra el bit CR1 está a 0, se saca un 1 a la salida. En ambos casos se pone a 1 durante 1 ciclo de reloj el OUTPUT_ENABLE.

4.5.4 Fifo To Output Register

Este módulo también es bastante importante dentro del receptor puesto que es el encargado de interactuar con la FIFO y de mandar los datos al registro de salida y al registro de estado del esclavo AXI. Su función es la de adquirir los posibles errores que viene del módulo del Line Receiver y la palabra de 32 bits y almacenarlo en la FIFO si la etiqueta de dicha palabra se encuentra dentro de los mapas de etiquetas. Por otro lado, también se encarga de actualizar el registro de estado constantemente si se produce algún cambio y el registro de salida si el procesador ha leído la palabra que se encontraba almacenada.

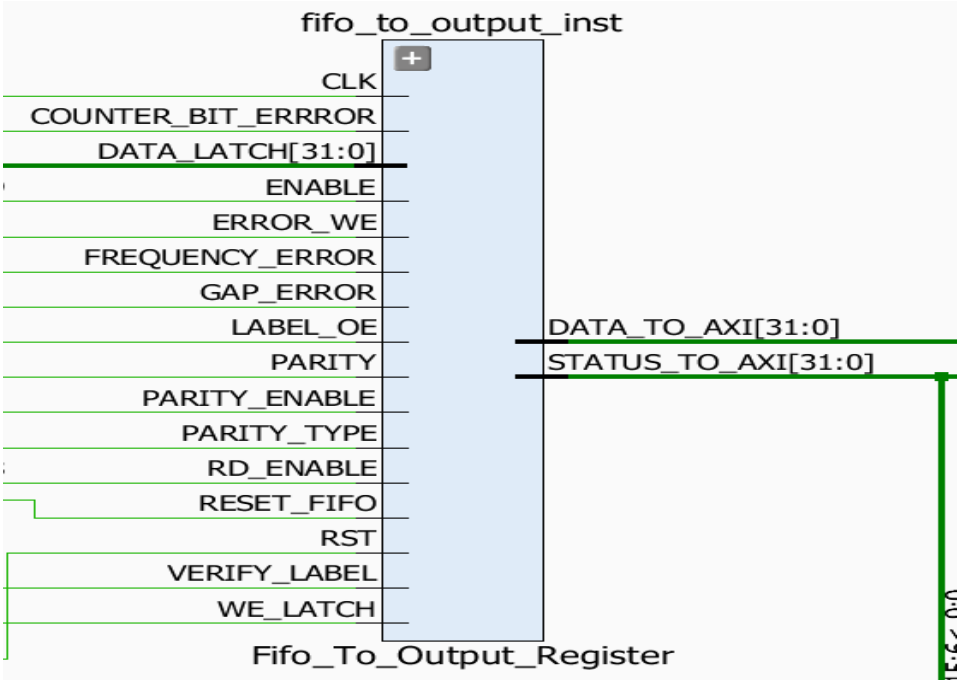


Ilustración 17. Esquemático del módulo Fifo To Output Register

Sus entradas son mayoritariamente las salidas de los módulos anteriormente mencionados como los posibles errores y la señal habilita la adquisición provenientes del Line Receiver, la palabra de 32 bits y la señal que habilita la adquisición viniendo del Latch, la verificación de la etiqueta y la señal que habilita cuando adquirirla que proviene del Label Check, los bits CR0, CR2, CR3 y CR25 definidos en la Tabla 2 que viene del Logic Control, la señal que viene del esclavo AXI e indica si se ha leído la palabra que se encontraba almacenada en el registro de salida, el reloj y el reset.

Internamente, el sistema en primer lugar se encarga de coger los posibles errores y la paridad cuando la señal de habilitar la captura se pone a 1 y comprueba si ha habido error de paridad o no según el valor de las entradas CR2 y CR3. Tras esto, almacena esos resultados. También espera a que llegue la señal que habilita la captura de la palabra de 32 bits y cuando llega, guarda la palabra. Cuando llega la señal que habilita la adquisición de la verificación de la etiqueta ,si la señal de verificación vale 1 envía los datos de los posibles errores y la palabra de 32 bits a la FIFO. Cabe destacar que el bit 32 de los datos que es el de la paridad se intercambia por el bit ST1.

Por otro lado, si el procesador indica al módulo que ya ha leído el dato almacenado, éste último se encarga de mandar una señal a la FIFO para que se envíe un nuevo dato al registro de salida y al registro de estado. Esto ocurre siempre que la FIFO no se encuentre vacía ya que si no, el módulo espera hasta que la FIFO le indique que no está vacía. Además el registro de estado aparte de actualizarse cuando se envía una nueva palabra se actualiza al instante cuando se produce un cambio en los bits ST6, ST7 y ST8. Se puede ver en detalle en la Tabla 3.

4.5.5 Logic Control

Este sistema está encargado de coger los datos del registro de control, separar los bits según se encuentran en la Tabla 2 (a excepción del gap) y enviarlos a los distintos módulos donde sean necesarios.

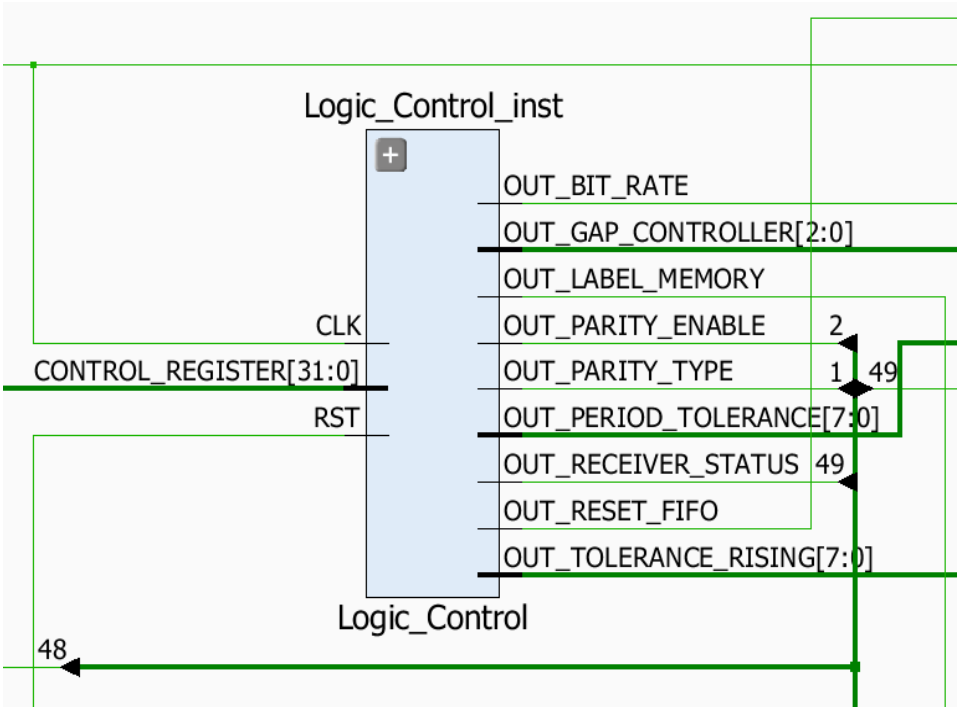


Ilustración 18. Esquemático del módulo Logic Control

Únicamente cuenta con 3 puertos de entrada que son el registro de control, el reloj y el reset. Debido a que separa el registro en las diferentes señales se tienen 9 puertos de salida que son el bit rate, el controlador de gap, la memoria de etiquetas (filtrar o no filtrarlas todas), el habilitador de paridad, el tipo de paridad, el periodo de tolerancia, el estado del receptor, el bit de reset de la FIFO y la tolerancia de subida.

Internamente el módulo se encarga de separar todo el registro de control en las diferentes señales que lo componen. Solo se produce una excepción para el caso del gap ya que en este caso, lo que hace el módulo es comprobar si el bit CR4 está a 1 o no. Si no lo está, se envía el periodo de bit por defecto que son cuatro periodos de bit. Si por contra sí que lo está, se envían los bits CR5-7.

4.5.6 FIFO

La FIFO utilizada en este proyecto es creada mediante la GUI citada anteriormente que puede ver en la Ilustración 7. Siguiendo los pasos del módulo FIFO Generator del catálogo de Xilinx obtendremos la FIFO que necesitamos.

El módulo FIFO es el módulo encargado de almacenar los datos provenientes del AXI para su posterior envío. Necesitaremos una FIFO con capacidad para 64 palabras de 64 bits cada una ya que en ella se almacenan las señales que se dirigirán al registro de salida y al de estado.

Las características de esta FIFO son muy concretas. El reset es activo en flanco de subida y permanece en estado de reset tres ciclos de reloj. Para escribir en ella cuenta con una entrada write_enable en la que al poner un 1 los datos que se encuentren en la entrada de datos de la FIFO serán almacenados en su registro interno durante ese mismo ciclo de reloj.

En la Ilustración 19 vemos que una vez activado el reset tienen que pasar tres flancos de subida de reloj hasta que la FIFO vuelve a aceptar señales de lectura y escritura.

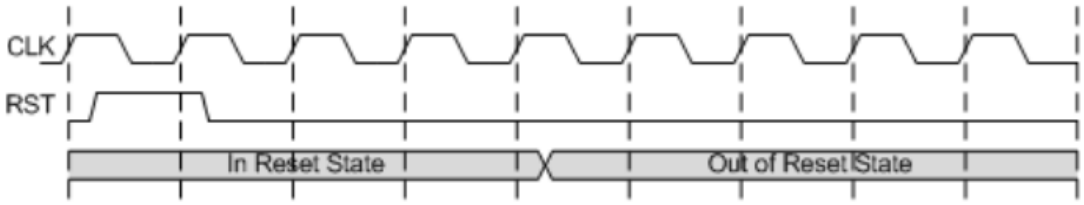


Ilustración 19. Visualización del funcionamiento de la FIFO (6)

Para leer datos, tiene otra entrada read_enable que puesta a 1 actualizará con el siguiente dato almacenado en su registro interno la salida de datos en el siguiente ciclo de reloj.

Además cuenta con salidas adicionales que proporcionan información sobre su estado. Una salida indicando si está llena y otra que indica si está vacía, fifo_full y fifo_empty. También cuenta con una salida que indica si es igual o mayor de 32 el número de datos que tiene almacenados. Es decir, prog_full indica si la FIFO está a la mitad o más de la mitad de su capacidad para así actualizar el bit del registro de estatus correspondiente.

La Ilustración 20 muestra el módulo FIFO con todas sus entradas y salidas como vemos la entrada y salida de datos es de 64 bits, ya que se almacenan en la misma FIFO los datos del registro de salida y los del registro de estado. La señal prog_full es de 1 bit. En dicha FIFO como mucho puede haber 64 palabras. Cuando la FIFO se llena, el bit de FIFO full se pone activo.

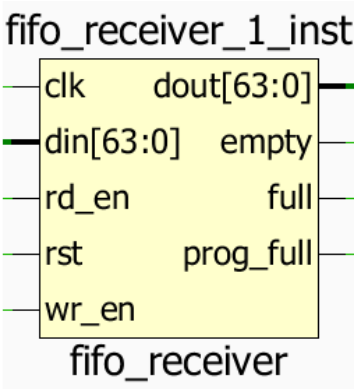


Ilustración 20. Módulo FIFO en formato caja negra

4.5.7 Esclavo AXI

Este módulo es el encargado de leer los datos del bus AXI que se dirigen al módulo receptor y actualizar sus registros de control y mapas de etiquetas en base a estos mensajes originados en el microprocesador. Por otro lado, cada vez que lee su registro de salida o datos, éste envía una señal de un ciclo de reloj de duración al Control Logic para hacerle saber que se ha leído esa palabra y así sacar una palabra almacenada en la FIFO.

Para la realización de este módulo nos hemos ayudado de la GUI de Vivado ya que nos proporciona una herramienta para generar automáticamente un módulo muy semejante al que necesitamos para este proyecto. Como ya vimos en la Ilustración 9 así comienza el proceso de creación del esclavo.

En la Ilustración 21 se muestra el segundo paso de la GUI para la creación de un esclavo AXI. Simplemente tenemos que añadir el nombre del nuevo IP y el sitio en el que queremos albergarlo.

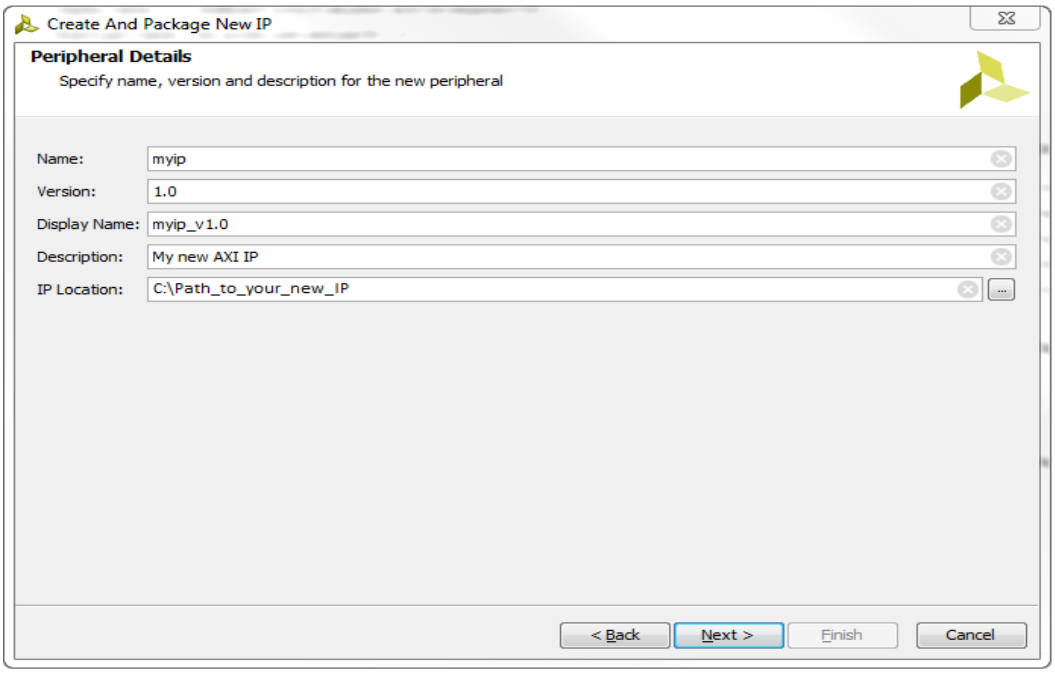


Ilustración 21. Selección del nombre y path del AXI esclavo

En la Ilustración 22 seleccionamos el tipo de interfaz AXI que necesitamos. En nuestro caso hemos seleccionado un interfaz AXI Lite esclavo con once registros de 32 bits y con capacidad para una sola palabra.

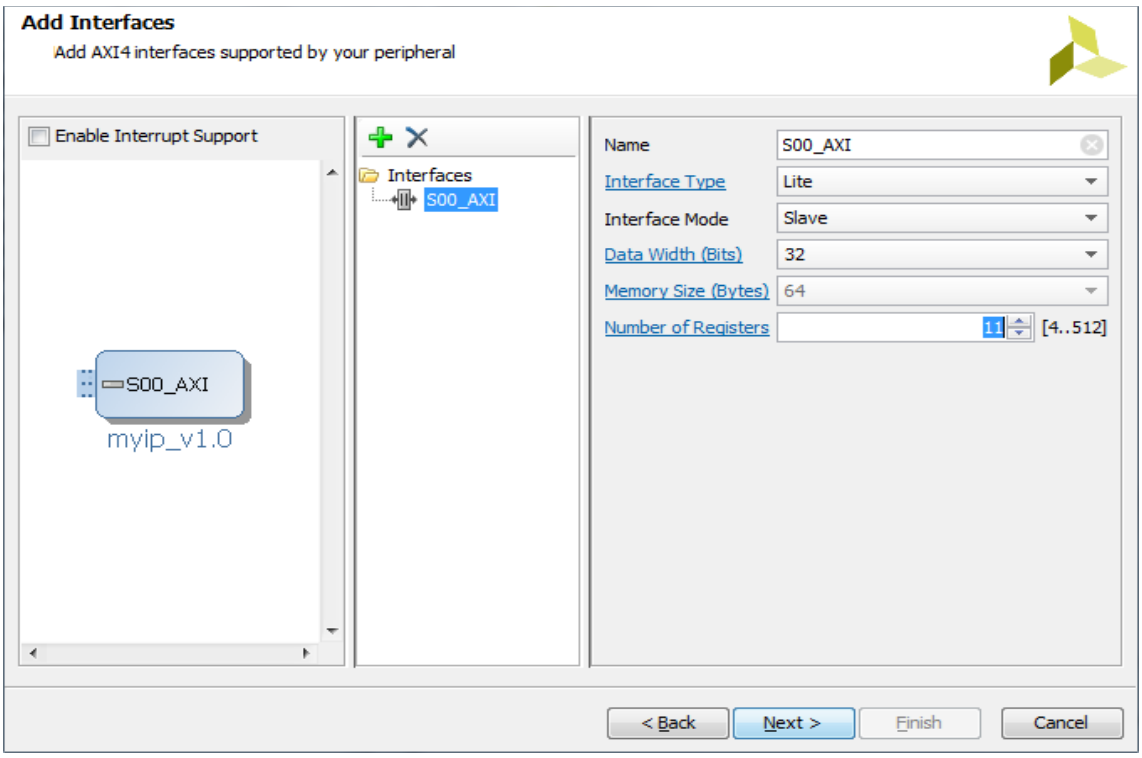


Ilustración 22. Definición del tipo de interfaz AXI a generar

En la siguiente ventana no tenemos más que seleccionar la opción de editar y ahí es cuando se nos creará un proyecto de edición del interfaz AXI con el bloque generado en código VHDL el cual podremos editar.

A este bloque le añadimos dos entradas y diez salidas. Las entradas estarán directamente asociadas al registro de estado y al registro de salida o datos de este módulo. Nueve de las diez salidas se corresponderán con los registros de control y los mapas de etiquetas del módulo. La décima salida, como ya hemos nombrado antes, será la encargada de indicar cuándo se han leído nuevos datos en el registro de datos del módulo, provenientes del microprocesador a través del bus AXI.

En la Ilustración 23 podemos ver como el módulo AXI se ha creado correctamente y mediante unas pequeñas modificaciones en el VHDL para añadir las entradas y salidas extras, resaltadas en azul, además de la lógica para que la salida OUT_REGISTER_READING funcione correctamente. Así nos queda el bloque lógico de la ilustración.

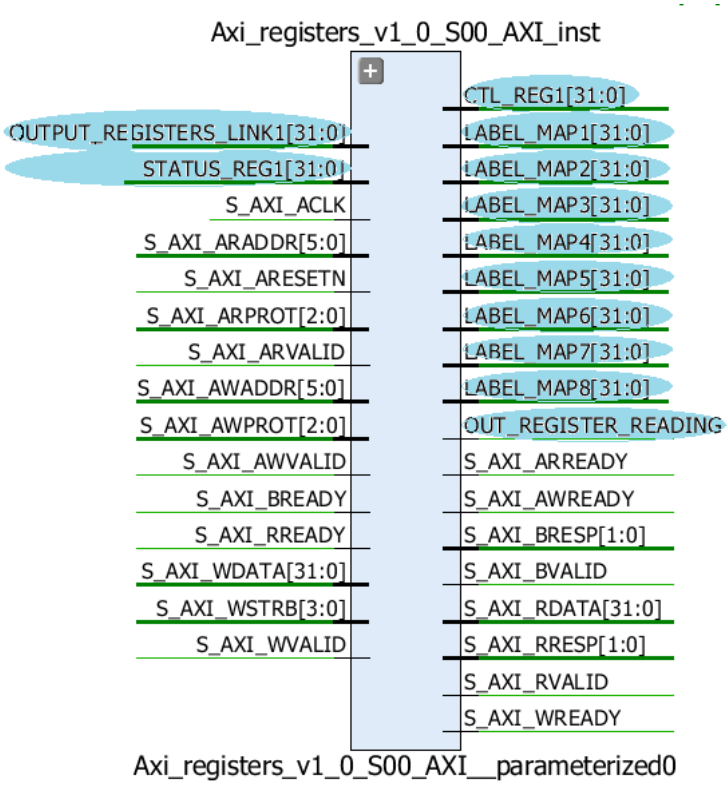


Ilustración 23. Módulo esclavo AXI personalizado en formato caja negra

En la Ilustración 24 vemos el esquemático de interconexión entre los registros del AXI y las salidas personalizadas añadidas a posteriori. La entrada del registro de estado no va a parar a un registro si no que va directamente a un multiplexor. Cuando el microprocesador pida los datos del estado del receptor se seleccionará esa entrada del multiplexor y los datos estarán listos para ser leídos directamente en la salida S_AXI_RDATA.

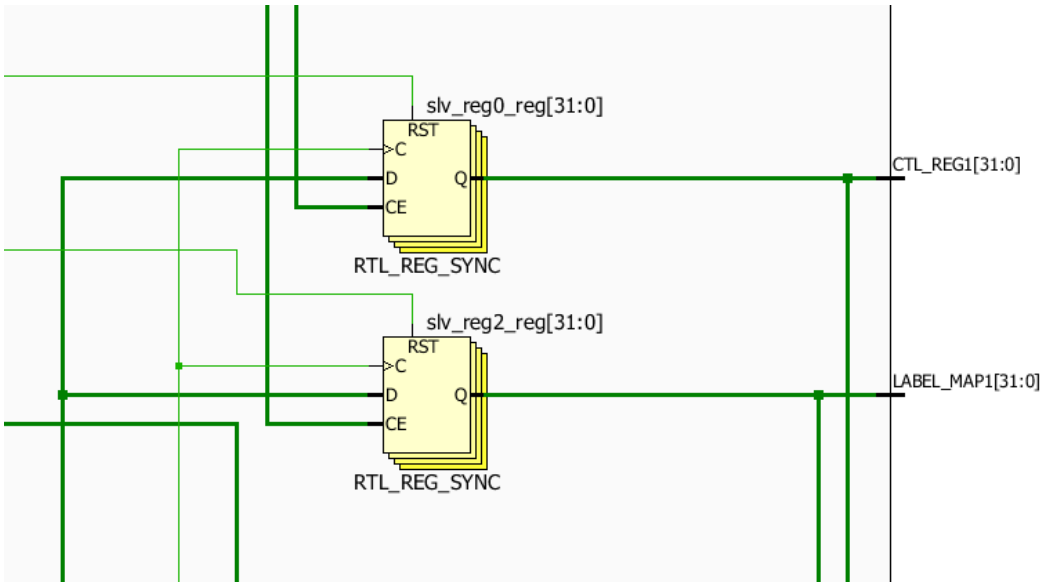


Ilustración 24. Interconexión interna entre registros y salidas adicionales

5. Diseño e Implementación de los Testbenches

Para la implementación de cada testbench automático, se generarán archivos mediante un script en Octave para los diferentes puertos de entrada y el testbench se encargará de coger los datos de esos ficheros y realizar con ellos las acciones correspondientes. A posteriori, para comprobar su correcto funcionamiento, dicho testbench generará unos ficheros con los datos de los puertos de salida y mediante otro script de Octave se comprobará que los datos de salida concuerdan con los ficheros de entrada teniendo en cuenta el comportamiento que produce el módulo de cada test.

5.1 Testbench de los Módulos del Receptor

En primer lugar se procede a describir el funcionamiento de los testbenches de cada módulo para finalmente explicar el testbench del receptor completo.

5.1.1 Testbench del Line Receiver

En este testbench lo que se hace es introducir ciertos ficheros que se especifican en la Tabla 4 con unos bits de entrada. Variando cada cierto tiempo los demás parámetros de entrada, cuando llega una palabra completa y el OUT_ENABLE se pone a 1 el testbench escribe en sus respectivos ficheros los valores que tienen los puertos de salida en ese instante. Cuando la simulación termina, se comprueba que el funcionamiento del módulo es correcto y que por tanto; no falla.

En la Tabla 4 se muestran a la izquierda los puertos y a la derecha los respectivos ficheros de lectura o escritura de datos.

Puerto	Archivo
RX1_HIGH y RX1_LOW	Rx_to_Line_Receiver.txt
IN_RISING_TOLERANCE	Rising_Tolerance_to_Line_Receiver.txt
IN_PERIOD_TOLERANCE	Period_Tolerance_to_Line_Receiver.txt
BIT_RATE	Bit_Rate_to_Line_Receiver.txt
GAP_CONTROLLER	Gap_Controller_to_Line_Receiver.txt
IN_RECEIVER_STATUS	Receiver_Status_to_Line_Receiver.txt
OUT_DATA	Output_Out_Data_from_Line_Receiver.txt
OUT_FREQUENCY_ERROR	Output_Counter_Bit_Error_from_Line_Receiver.txt
OUT_GAP_ERROR	Output_Gap_Error_from_Line_Receiver.txt
OUT_PARITY_VERIFY	Output_Parity_Verify_from_Line_Receiver.txt
OUT_ENABLE	Output_Enable_from_Line_Receiver.txt

Tabla 4. Mapeo de puertos del Line Receiver con sus respectivos ficheros

En la Ilustración 25 se puede apreciar un fragmento del total de la simulación del Line Receiver.

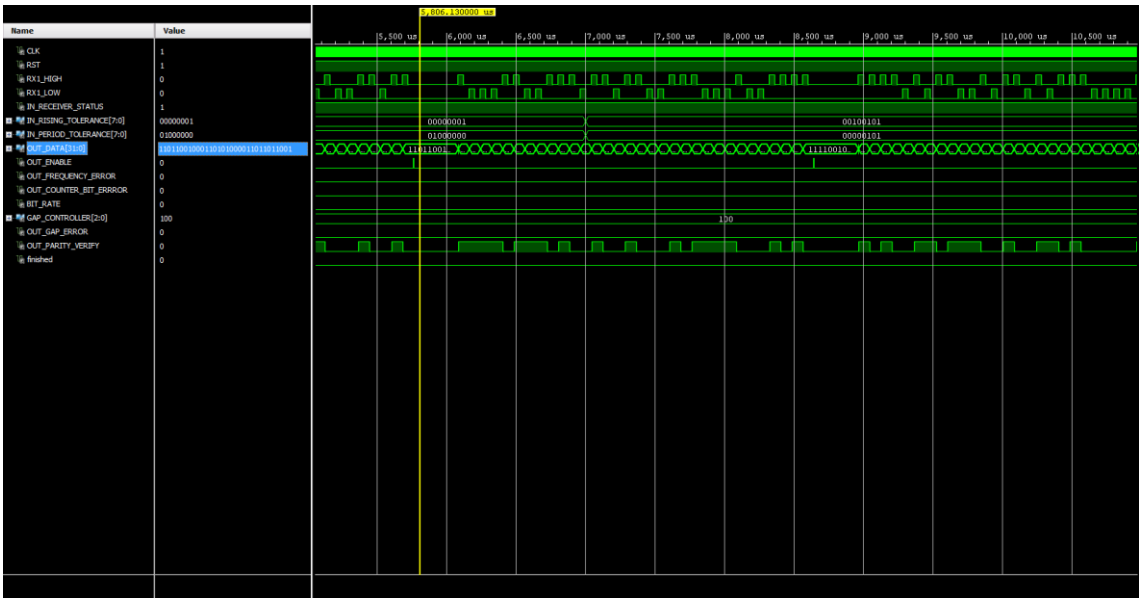


Ilustración 25. Simulación del testbench automático del módulo Line Receiver

Para comprobar que todo funciona correctamente se utiliza un script de Octave que almacena los bits agrupados de 32 en 32, generando palabras. El script además calcula la paridad y genera los posibles errores si éstos se producen con los datos de entrada que se disponen. Todo lo mencionado anteriormente se almacenan en variables. Una vez finalizado el script, comprueba que los ficheros de salida creados por el testbench son iguales que las variables calculadas con el script.

5.1.2 Testbench del Latch

En este testbench lo que se hace es introducir ciertos ficheros que se especifican en la Tabla 5. Una de cada 3 palabras que entran por el puerto DATA_LATCH pone el bit del WE_LATCH a 1 durante 1 ciclo de reloj. Cuando el OUT_ENABLE se pone a 1, el testbench escribe en su respectivo fichero el valor del puerto de salida OUT_DATA. Cuando la simulación termina, se comprueba que el funcionamiento del módulo es correcto y que por tanto; no falla.

En la Tabla 5 se muestran a la izquierda los puertos y a la derecha los respectivos ficheros de lectura o escritura de datos.

Puerto	Archivo
DATA_LATCH	Input_data_to_Latch.txt
WE_LATCH	Input_data_to_Latch.txt
OUT_DATA	Output_data_from_Latch.txt
OUT_ENABLE	Output_Enable_from_Latch.txt

Tabla 5. Mapeo de puertos del Latch con sus respectivos ficheros

En la Ilustración 26 se puede ver un fragmento de toda la simulación y como la etiqueta ha sido volteada en el puerto de salida de datos.

Ilustración 26. Simulación del testbench automático del módulo Latch

Para comprobar que lo anteriormente mencionado funciona como debiera, se utiliza un script de Octave que introduce las palabras del fichero de entrada "Input_data_to_Latch.txt" en una variable. En dicha variable se le da la vuelta a los 8 últimos bits que corresponden a las etiquetas. Finalmente, se comprueba si las palabras de la variable son iguales a las palabras del fichero de salida "Output_data_from_Latch.txt".

5.1.3 Testbench del Label Check

En este testbench lo que se hace es introducir ciertos ficheros que se especifican en la Tabla 6. Una de cada 3 palabras que entran por el puerto IN_LABEL_CHECK pone el bit del WE_LATCH a 1 durante un ciclo de reloj. Además se varía cada cierto tiempo los mapas de etiquetas y el puerto LABEL_ENABLE . Cuando el OUTPUT_ENABLE se pone a 1, el testbench escribe en sus respectivos ficheros los valores que tienen los puertos de salida en ese instante. Cuando la simulación termina, se comprueba que el funcionamiento del módulo es correcto y que por tanto; no falla.

En la Tabla 6 se muestran a la izquierda los puertos y a la derecha los respectivos ficheros de lectura o escritura de datos.

Puerto	Archivo
IN_LABEL_CHECK	Label_Check_to_LabelCheck.txt
IN_LABEL_MAP1	Label_Map1_to_LabelCheck.txt
IN_LABEL_MAP2	Label_Map2_to_LabelCheck.txt
IN_LABEL_MAP3	Label_Map3_to_LabelCheck.txt
IN_LABEL_MAP4	Label_Map4_to_LabelCheck.txt
IN_LABEL_MAP5	Label_Map5_to_LabelCheck.txt
IN_LABEL_MAP6	Label_Map6_to_LabelCheck.txt
IN_LABEL_MAP7	Label_Map7_to_LabelCheck.txt
IN_LABEL_MAP8	Label_Map8_to_LabelCheck.txt
LABEL_ENABLE	Label_Enable_to_LabelCheck.txt
WE_ENABLE	Label_Enable_to_LabelCheck.txt
VERIFY_LABEL	Output_LabelVerify_from_LabelCheck.txt
OUTPUT_ENABLE	Output_Enable_from_Label_Check.txt

Tabla 6. Mapeo de puertos del Label Check con sus respectivos ficheros

En la Ilustración 27 se puede ver un fragmento de toda la simulación y como saca un 1 si la etiqueta está dentro del mapa y un 0 si no lo está.

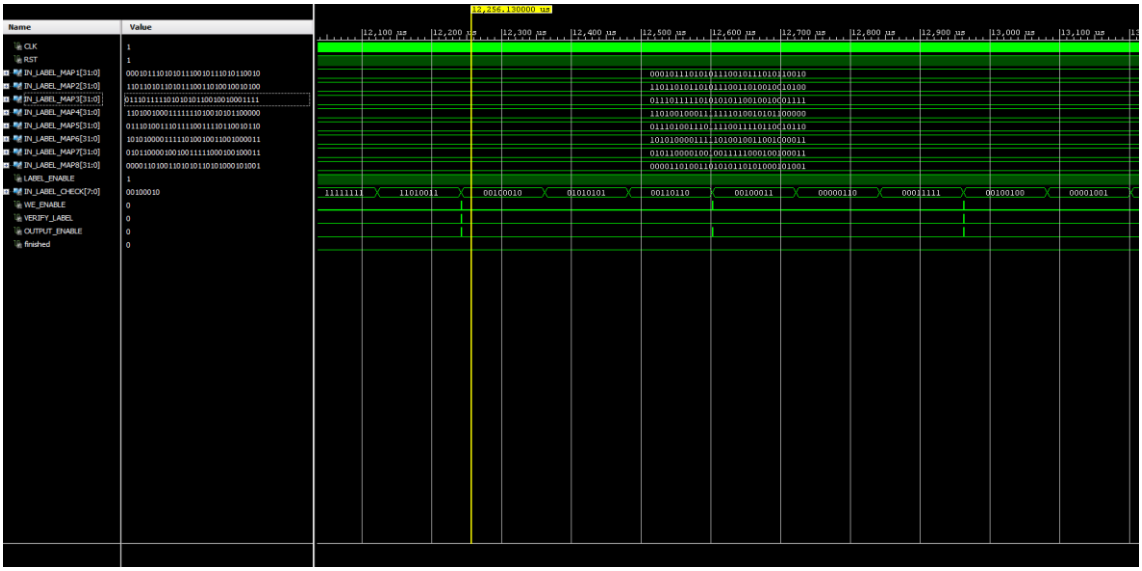


Ilustración 27. Simulación del testbench automático del módulo Label Check

Para comprobar que lo anteriormente mencionado funciona correctamente, se utiliza un script de Octave que va cogiendo cada vez una etiqueta del fichero "Label_Check_to_LabelCheck.txt". Esta etiqueta binaria se transforma a un valor entero y se introduce como posición de un vector que contiene los mapas de etiquetas. El valor de esta posición se almacenan en una matriz y tras ejecución del script se verifica si esos resultados son iguales que el fichero "Output_LabelVerify_from_LabelCheck.txt"

5.1.4 Testbench del Fifo To Output Register

En este testbench lo que se hace es introducir ciertos ficheros que se especifican en la Tabla 7. Este testbench debido a que se envían señales del procesador cada poco tiempo fuerza a que en la FIFO se esté continuamente introduciendo una palabra y vaciándose y por tanto; a que el registro de estado vaya cambiando constantemente por el bit ST7. Cada vez que las salidas cambian de valor, se escriben en los correspondientes ficheros. Cuando la simulación termina, se comprueba que el funcionamiento del módulo es correcto y que por tanto; no falla.

En la Tabla 7 se muestran a la izquierda los puertos y a la derecha los respectivos ficheros de lectura o escritura de datos.

Puerto	Archivo
ENABLE	Status_Enable_to_Fifo_To_Output_Register.txt
RD_ENABLE	Axi_Enable_to_Fifo_To_Output_Register.txt
RESET_FIFO	Reset_Fifo_to_Fifo_To_Output_Register.txt
DATA_LATCH	Data_to_Fifo_To_Output_Register.txt
WE_LATCH	Data_to_Fifo_To_Output_Register.txt
VERIFY_LABEL	Verify_Label_to_Fifo_To_Output_Register.txt
LABEL_OE	Label_Oe_to_Fifo_To_Output_Register.txt
FREQUENCY_ERROR	Frequency_Error_to_Fifo_To_Output_Register.txt
COUNTER_BIT_ERROR	Counter_Bit_Error_to_Fifo_To_Output_Register.txt
GAP_ERROR	Gap_Error_to_Fifo_To_Output_Register.txt
PARITY	Parity_to_Fifo_To_Output_Register.txt
ERROR_WE	Error_We_to_Fifo_To_Output_Register.txt
PARITY_ENABLE	Parity_Enable_to_Fifo_To_Output_Register.txt
PARITY_TYPE	Parity_Type_to_Fifo_To_Output_Register.txt
DATA_TO_AXI	Output_Out_Data_from_Fifo_To_Output_Register.txt
STATUS_TO_AXI	Output_Out_Status_from_Fifo_To_Output_Register.txt

Tabla 7. Mapeo de puertos del Fifo To Output Register con sus respectivos ficheros

En la Ilustración 28 se puede ver toda la simulación, como se almacenan datos en la FIFO si la verificación de la etiqueta es correcta, como se actualizan los datos de salida cada vez que el procesador indica que se ha leído datos y como el registro de estado se actualiza constantemente.

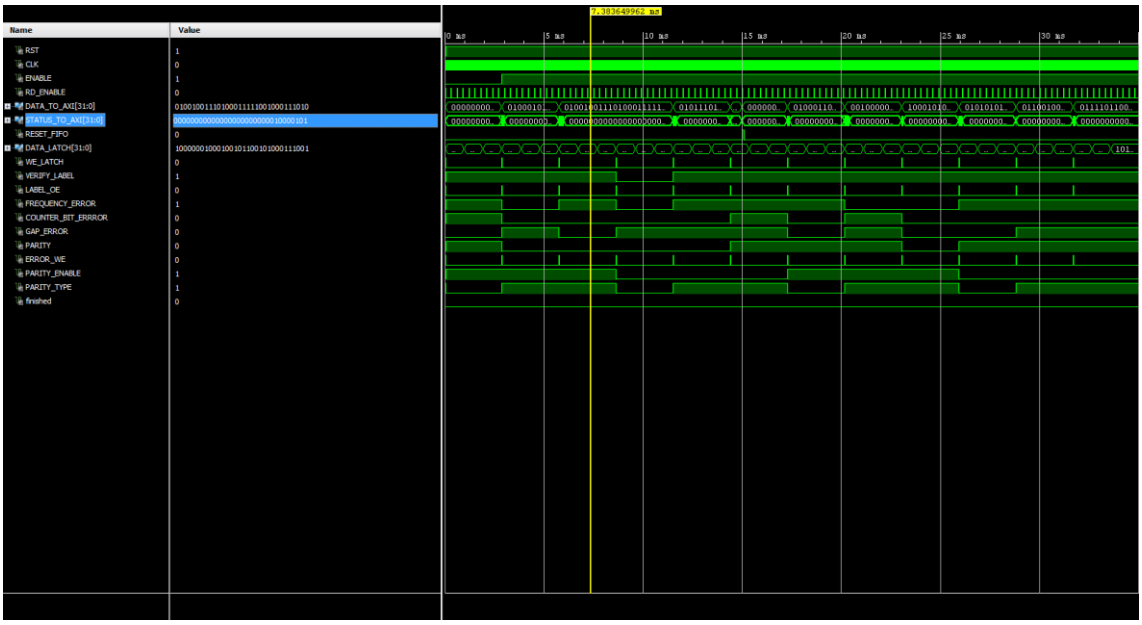


Ilustración 28. Simulación del testbench automático del módulo Fifo To Output Register

Para comprobar que lo anteriormente mencionado funciona correctamente, se utiliza un script de Octave que almacena en una variable todos los datos de entrada. Con cada palabra del fichero de entrada "Data_to_Fifo_To_Output_Register.txt" se comprueba si la verificación de la etiqueta, que se adquiere a la vez que cada dato del fichero de entrada y se coge del archivo "Verify_Label_to_Fifo_To_Output_Register.txt", está a 1. En ese caso, esa palabra se almacena en otra variable que haría de FIFO. Esto se repite cada vez que se coge una palabra. Cuando un dato proveniente del "Axi_Enable_to_Fifo_To_Output_Register.txt " es 1, se traslada una palabra almacenada en esa variable que hace de FIFO a una variable que guarda los datos de salida. El mismo proceso se efectúa para los datos que salen al registro de estado pero con los datos de entrada de los errores, solo que además se escribe en la variable de salida de estado, bien cuando se produce un cambio porque la FIFO está vacía o llena, o porque varía el bit data_ready del registro de estado. Finalmente, se comprueba si esas variables de salida son iguales que los ficheros de salida generados en la simulación.

5.1.5 Testbench del Logic Control

En este testbench lo que se hace es introducir ciertos ficheros que se especifican en la Tabla 8. Cada vez que se introduce un nuevo dato del "Input_data_to_LogicControl.txt", las salidas escriben sus datos en los respectivos ficheros de salida. Cuando la simulación termina, se comprueba que el funcionamiento del módulo es correcto y que por tanto; no falla.

En la Tabla 8 se muestran a la izquierda los puertos y a la derecha los respectivos ficheros de lectura o escritura de datos.

Puerto	Archivo
CONTROL_REGISTER	Input_data_to_LogicControl.txt
OUT_BIT_RATE	Output_BitRate_from_LogicControl.txt

OUT_RECEIVER_STATUS	Output_Status_from_LogicControl.txt
OUT_LABEL_MEMORY	Output_LabelMemory_from_LogicControl.txt
OUT_PARITY_TYPE	Output_ParityType_from_LogicControl.txt
OUT_PARITY_ENABLE	Output_ParityEnable_from_LogicControl.txt
OUT_GAP_CONTROLLER	Output_GapController_from_LogicControl.txt
OUT_TOLERANCE_RISING	Output_ToleranceRising_from_LogicControl.txt
OUT_PERIOD_TOLERANCE	Output_PeriodTolerance_from_LogicControl.txt
OUT_RESET_FIFO	Output_ResetFifo_from_LogicControl.txt

Tabla 8. Mapeo de puertos del Logic Control con sus respectivos ficheros

En la Ilustración 29 se puede ver un fragmento de toda la simulación. También se puede apreciar como los datos de entrada del CONTROL_REGISTER se diseminan en una gran variedad de datos de salida.

Ilustración 29. Simulación del testbench automático del módulo Logic Control

Para comprobar que lo anteriormente mencionado funciona correctamente, se utiliza un script de Octave que verifica si los bits de los diferentes puertos de salida, que han sido almacenados en sus respectivos ficheros, concuerdan con los bits a los que corresponden dentro de cada palabra del fichero " Input_data_to_LogicControl.txt ".

5.2 Testbench del Receptor Completo

Finalmente, se pasa a explicar el funcionamiento del testbench del receptor completo. Este test es el más complejo ya que engloba a todos los módulos. Los diferentes ficheros bien sean de lectura o escritura se especifican en la Tabla 9. Para comprobar que el sistema funciona como es debido, el testbench introducirá por los puertos RX1_HIGH y RX1_LOW bits, que al agruparse en palabras, estén la mayoría de las etiquetas. También se introducirán 1's en ambos puertos a la vez, que provocan errores de contador de bits y un número suficiente

de bits al azar para llenar la FIFO. Además se introducirán en los mapas de etiquetas y registro de control, datos que irán variando cada un cierto tiempo especificado en el testbench. Cada vez que salga un dato diferente por los puertos de salida se escribe en sus respectivos ficheros. Cuando la simulación termina, se comprueba que el funcionamiento del módulo es correcto y que por tanto; no falla.

En la Tabla 9 se muestran a la izquierda los puertos y a la derecha los respectivos ficheros de lectura o escritura de datos.

Puerto	Archivo
RX1_HIGH	Rx_to_Receiver.txt
RX1_LOW	Rx_to_Receiver.txt
CTL_REG1	Control_Register_to_Receiver.txt
LABEL_MAP1	Label_Map1_to_Receiver.txt
LABEL_MAP2	Label_Map2_to_ Receiver.txt
LABEL_MAP3	Label_Map3_to_ Receiver.txt
LABEL_MAP4	Label_Map4_to_ Receiver.txt
LABEL_MAP5	Label_Map5_to_ Receiver.txt
LABEL_MAP6	Label_Map6_to_ Receiver.txt
LABEL_MAP7	Label_Map7_to_ Receiver.txt
LABEL_MAP8	Label_Map8_to_ Receiver.txt
INPUT_REGISTER_READING	Register_Reading_to_Receiver.txt
OUTPUT_REGISTERS_LINK1	Out_Data_from_Receiver.txt
STATUS_REG1	Out_Status_from_Receiver.txt

Tabla 9. Mapeo de puertos del Receptor con sus respectivos ficheros

En la Ilustración 30 se puede ver un fragmento de toda la simulación. Además se puede apreciar como cada vez que la parte del procesador indica que ya ha leído el dato almacenado en el registro de salida se saca un nuevo dato de la FIFO y como se actualiza el registro de estado.

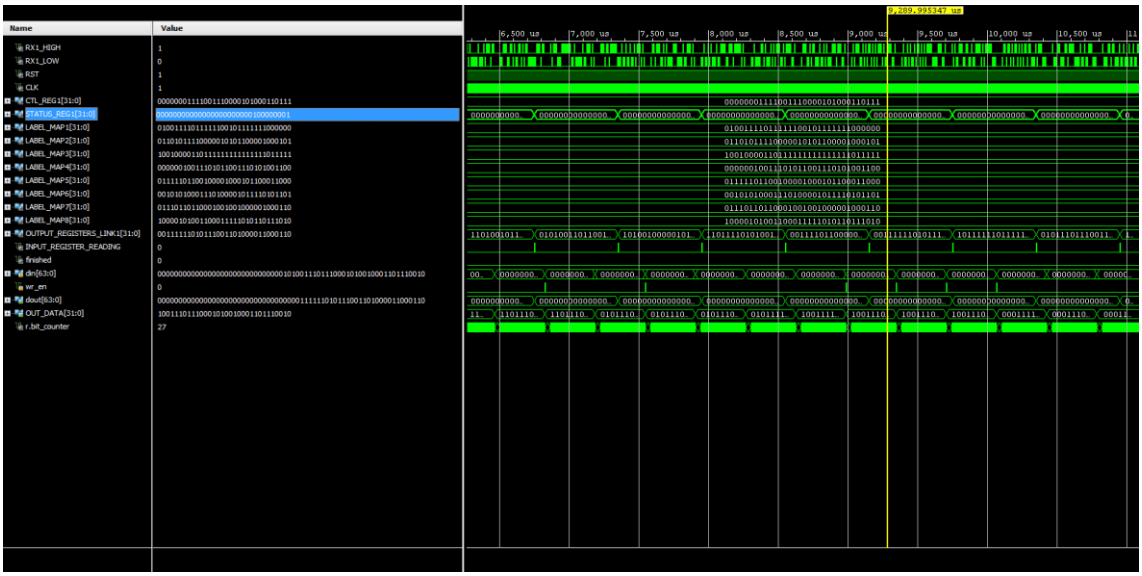


Ilustración 30. Simulación del testbench automático del Receptor completo

Para comprobar que el receptor funciona correctamente, se utiliza un script de Octave. Este script es el único script de todos los realizados para la comprobación de los testbenches que es temporal; es decir que hace un bucle for que va recorriendo todos los ficheros de entrada con una variable temporal que se incrementa cada 20 ns. Cada vez que la variable temporal iguala o supera el tiempo en el que finaliza un dato de un fichero de entrada, pasa al dato siguiente. De esta manera se consigue simular el funcionamiento interno del receptor completo con un script, pero con la diferencia que aquí toma los ficheros de entrada y genera y almacena en variables los datos de del registro de salida y del registro de estado sin alteraciones temporales del reloj, cosa que si puede ocurrir en las simulaciones de Vivado. Por tanto; si los ficheros " Out_Data_from_Receiver.txt " y "Out_Status_from_Receiver.txt" son iguales a las variables de datos del registro de salida y del registro de estado se verifica que el testbench es correcto y que el sistema funciona a la perfección.

6. Síntesis e Implementación

Tras la comprobación del correcto funcionamiento tanto del receptor completo como de los diferentes módulos que lo componen, el siguiente paso es sintetizar e implementar el código.

La síntesis es el proceso por el cual se transforma el modelo el código en VHDL a un nivel de abstracción inferior o lo que es lo mismo, convierte el código a nivel de puertas lógicas y cables.

En la Ilustración 31 se puede apreciar cómo tras realizar la síntesis el código VHDL se convierte a nivel lógico.

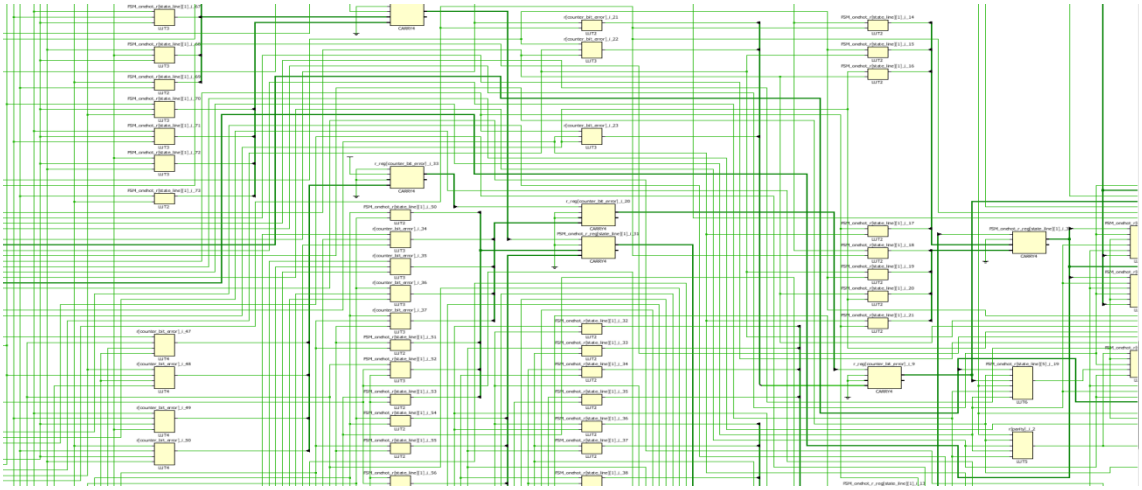


Ilustración 31. Esquemático de un módulo sintetizado

La implementación es el proceso por el cual se pasa a distribuir el diseño en la lógica programable de la FPGA en cuestión a utilizar. Por tanto; aquí sí que toma gran relevancia el modelo de la placa que se va a utilizar ya que variará la implementación en función de cual se utilice.

En la Ilustración 32 se muestra el diseño del dispositivo tras realizar la implementación. Los bloques subrayados en azul son los bloques que se utilizan e interconexionan para realizar la función que se ha especificado en el código.

Ilustración 32. Implementación de un módulo en el dispositivo

Hay que destacar que tras realizar la simulación de comportamiento de los diferentes módulos es necesario realizar las simulaciones postsíntesis y postimplementación ya que aquí se tienen en cuenta los retrasos físicos tanto del reloj como de las diferentes señales de entrada. Esto hace, que se pueda no obtener tras este tipo de simulaciones el resultado deseado y que haya que modificar el código.

7. Programación de la FPGA

Para esta tarea hemos decidido usar una Zedboard debido a su disponibilidad y a contar con un chip (Zynq 7000) que contiene una parte programable (PL o Programmable Logic) y otra parte de procesador (PS o Processing System). A este tipo de chips se los denomina SoC (System on chip) programable y nos va a permitir probar nuestro hardware corriendo una aplicación de prueba que interactúe con él.

Mediante el SDK de Xilinx es posible introducir un software simple que probara el módulo receptor junto con un módulo transmisor ARINC 429 cuya implementación se realizó de manera paralela.

El proceso de programación es sencillo, solo necesitamos unos drivers simples para poder interactuar con el transmisor y receptor y un pequeño programa de prueba. Este software se ha realizado en C y lo único que hace es escribir palabras de 4 bytes en el registro de datos del esclavo AXI del transmisor. A su vez el software recoge las palabras que llegan al receptor y las muestra en un terminal conectado al bus serie de la FPGA. Esta transmisión se realiza manteniendo el registro de control fijo a un valor que nos permite transmitir.

Para llegar a este punto de programación software hemos tenido que añadir unos módulos adicionales en nuestro diseño. Como se puede ver en la Ilustración 33 estos módulos son los que hacen referencia al PS. Hemos añadido los bloques correspondientes al bus AXI, al procesador (procesador y resets del sistema) y a los dos puertos UART para programar la FPGA y monitorizar y continuar con el proyecto, además se ha añadido un último módulo para depurar la aplicación mientras esté corriendo. Los dos módulos que quedan por nombrar son el transmisor y receptor ARINC 429.

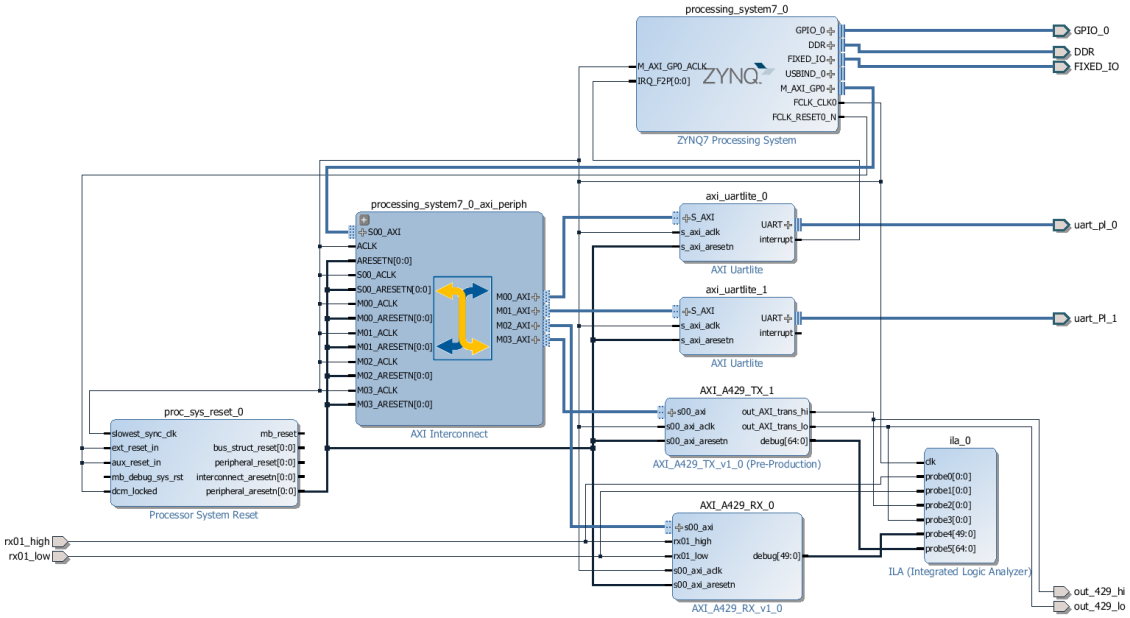


Ilustración 33. Diagrama de bloques del sistema completo

Para interconectar el transmisor y receptor en la FPGA les hemos asignado unos puertos de salida y entrada respectivamente. Como vemos en la Ilustración 34 cortocircuitamos los puertos del transmisor y receptor mediante el cable marrón y el cable verde. Con la FPGA conectada como muestra la Ilustración 34 y el diseño de bloques como en la Ilustración 33 ya está todo listo para realizar la programación.

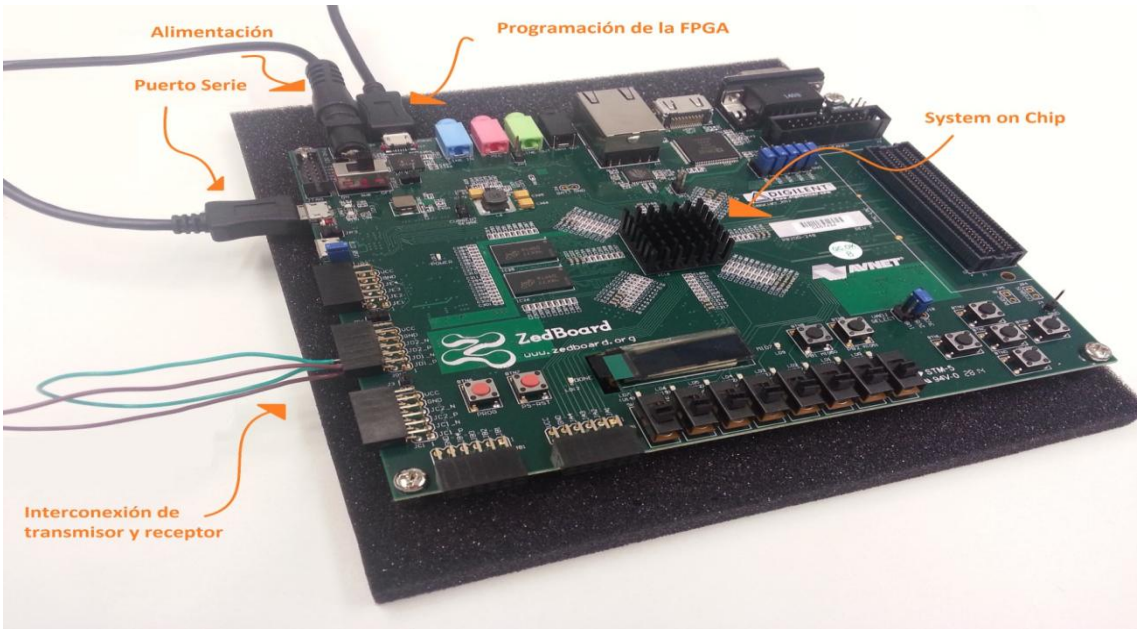


Ilustración 34. Zedboard con transmisor y receptor

La Ilustración 35 nos muestra el interfaz del SDK de Xilinx con el que una vez exportado el bitstream desde Vivado nos permite programar la FPGA y correr una aplicación en el PS. También vemos indicados el botón con el que programar la FPGA y el terminal en el que veremos los datos del bus serie. Para poder hacer esto el terminal deberá estar configurado a una tasa de baudios de 115200.

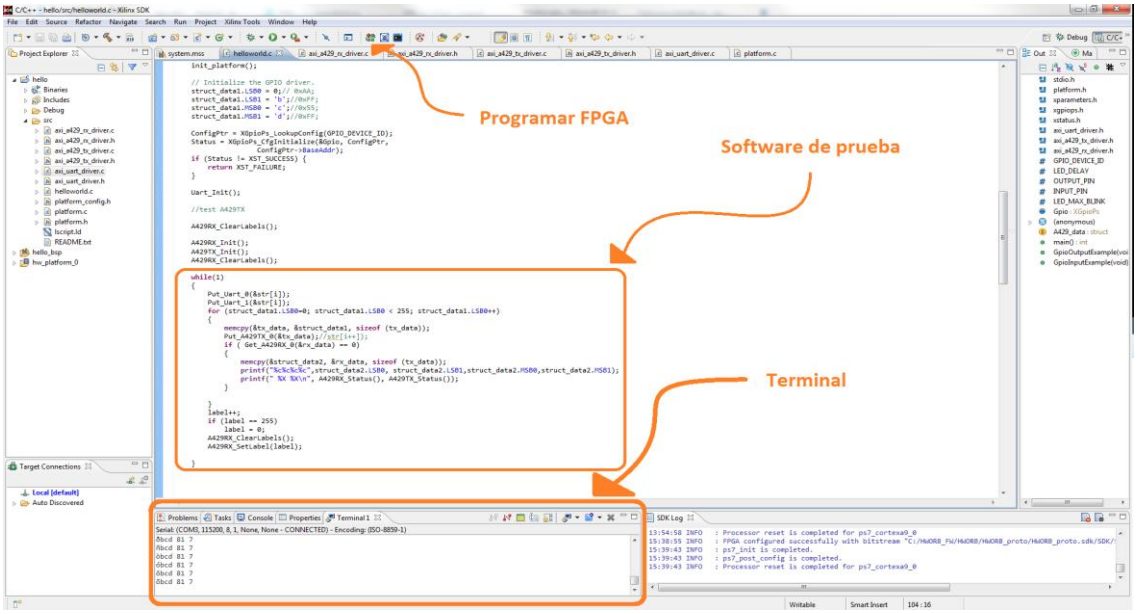


Ilustración 35. Interfaz del SDK de Xilinx

En la parte destacada del software podemos ver cómo lo único que hacemos es escribir en el registro de datos del transmisor y, si han llegado datos al receptor, los mostramos por el bus junto con los registro de estatus del receptor y transmisor en ese orden. Los 8 bits LSB de los datos escritos van cambiando en cada iteración recorriendo de esta manera todas las posibles etiquetas, que es por lo que vemos en el terminal como va cambiando la primera letra de la palabra recorriendo todo el alfabeto ASCII.

La Ilustración 36 muestra como se depura mediante el ILA (Integrated Logic Analizer). Esta herramienta nos permite ver las señales internas de la placa en tiempo real para en última instancia ver porque están fallando los módulos, en caso de que fallen.

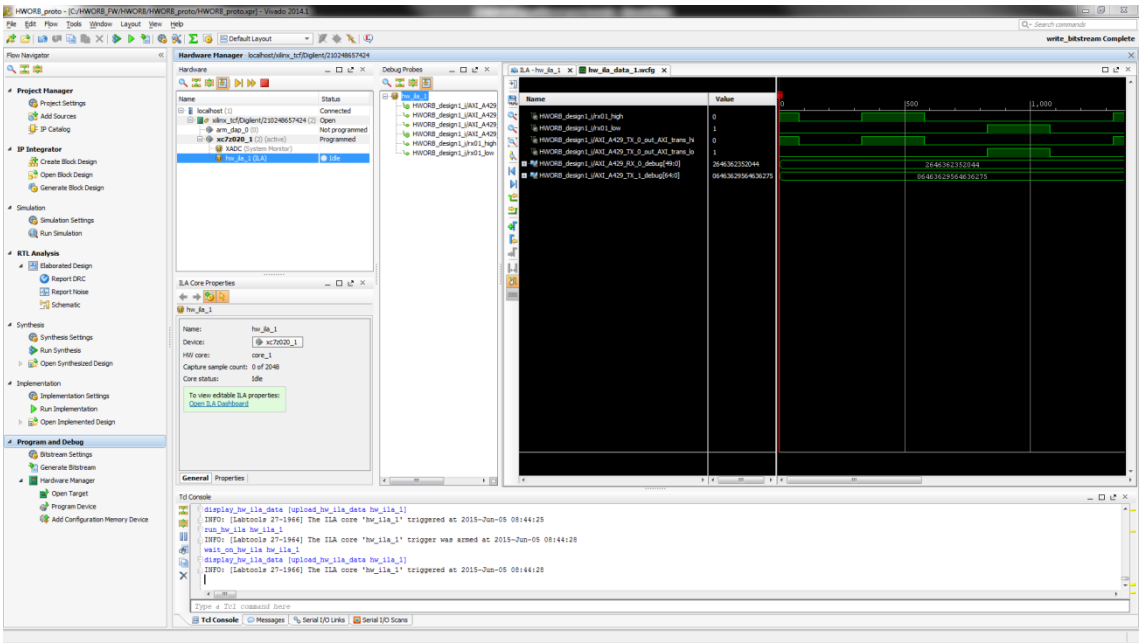


Ilustración 36. ILA integrado en Vivado en ejecución

8. Análisis de Resultados y Conclusiones

Como principal resultado se puede comentar, que se ha conseguido cumplir todos los objetivos propuestos al principio del proyecto. Es decir, se ha conseguido desarrollar un receptor ARINC 429, hacer test tanto básicos como automáticos para simular los módulos que forman el receptor y el receptor completo, y así poder tener a gran escala una primera impresión de si funciona lo que se ha diseñado o no y por último, se ha conseguido comprobar mediante scripts en Octave, que todos los módulos y el receptor completo funcionan a la perfección. Como guinda del pastel, se ha programado en una Zedboard y se ha verificado que se reciben datos perfectamente.

Además, de este proyecto se puede extraer que realizar cualquier implementación de hardware crítico requiere una cantidad de simulaciones, pruebas y documentación muy alta, lo que se traduce en un gasto importante de tiempo. A pesar de ello el resultado final es de una calidad que compensa con creces ese gasto.

También podemos ver como la tecnología y protocolos utilizados en aviación son muy concretos y la información sobre ellos se limita a una visión generalista. Por tanto la búsqueda de información se traduce en una larga tarea que te hace recurrir a los documentos de definición del protocolo.

VHDL es un lenguaje muy amplio del cual es sintetizable un porcentaje bastante bajo y aun utilizando ese pequeño porcentaje, al realizar las simulaciones funcionales y temporales el módulo puede llegar a fallar por diversas causas. Esto produce una continua búsqueda del código que mejor encaje con el comportamiento que necesitamos implementar, así como un análisis de las diferentes sintetizaciones posibles.

Pero hay que destacar que VHDL a pesar de crear quebraderos de cabeza también hace que el diseño de hardware complejo sea mucho más sencillo. Permitiendo abstraerte de la electrónica durante la mayor parte del proceso de desarrollo.

Las herramientas que proporciona Xilinx te permiten una visualización de los componentes y las conexiones además de proporcionarte ayuda a la hora de configurar los puertos de la FPGA. También cuenta con un sintetizador muy potente capaz de traducir tu módulo VHDL en una gran cantidad de FPGAs del mercado.

Estas herramientas necesitan de licencia para ser usadas y, aunque la licencia gratuita te permite hacer gran cantidad de cosas solo puedes usarla para un ordenador. Esta licencia suele entrar en conflicto con el antivirus lo que te obliga a desconectarlo temporalmente mientras Vivado este activo ya que de lo contrario te encontraras con la difícil tarea de conseguir otra. Además Vivado experimenta múltiples fallos repentinos que te obligan a volver a ejecutarlo con la implícita pérdida irremediable de datos y tiempo.

A la hora de ejecutar una simulación podemos usar el simulador Xsim integrado con Vivado o el simulador Modelsim que requiere de licencia. El simulador Xsim es funcional y permite ejecutar cualquier simulación pero cuando hablamos como en este proyecto de amplias simulaciones que llegan a durar días el Xsim puede llegar a ser excesivamente lento además de llegar a llenar la RAM impidiendo finalizar la simulación y provocando fallos en Vivado. Por eso se hace necesario emplear un simulador como Modelsim que no sobrecarga la RAM de nuestro ordenador y es ligeramente más rápido.

Durante el transcurso del proyecto se hizo claro que la Zedboard fue un gran acierto al incorporar un System on Chip que nos ha permitido probar el funcionamiento del hardware una vez implementado en su parte de lógica programable (PL) mediante una pequeña aplicación corriendo en su procesador basado en ARM (PS).

Finalmente concluimos que los objetivos de este proyecto han estado bien definidos y estructurados ya que se ha terminado con éxito cumpliendo todos ellos y en el rango de tiempo estipulado.

9. Líneas Futuras

La continuación lógica de este proyecto es acoplar un controlador de línea como uno de los mencionados en el apartado 4.2 Definición Técnica de un Receptor ARINC 429. Además de, una vez hecho esto, probar la comunicación de nuestro módulo con un dispositivo real que emplee esta tecnología de comunicación.

Una vez realizado el apartado hardware este proyecto tiene un gran rango de ampliación en su software, ya que la aplicación de prueba realizada para este es muy básica. Se quiere integrar un sistema linux en la Zedboard que nos permita correr aplicaciones propias sin necesidad de conectarla a un ordenador. Pudiendo así conectar directamente un monitor y un teclado a la placa.

Además como hemos hecho referencia en la Introducción esta es una pequeña parte de un proyecto mucho mayor. Por tanto el siguiente paso en este proyecto será la implementación de módulos transmisor y receptor AFDX y 1553.

Y una vez elaborados todos los protocolos necesarios para un sistema de comunicación de aviónica, el siguiente paso sería el aislamiento de los IP Cores. Es decir asignar a cada IP Core implementado un espacio delimitado por lógica sin utilizar dentro de la FPGA. Estos bloques aislados cuentan con entradas y salidas para interactuar con el resto de IP Cores.

En la Ilustración 37 podemos ver el significado de aislar un IP Core de manera gráfica. Como vemos quedan espacios libres en la FPGA pero este paso no es más que un medio para llegar a la reconfiguración parcial.

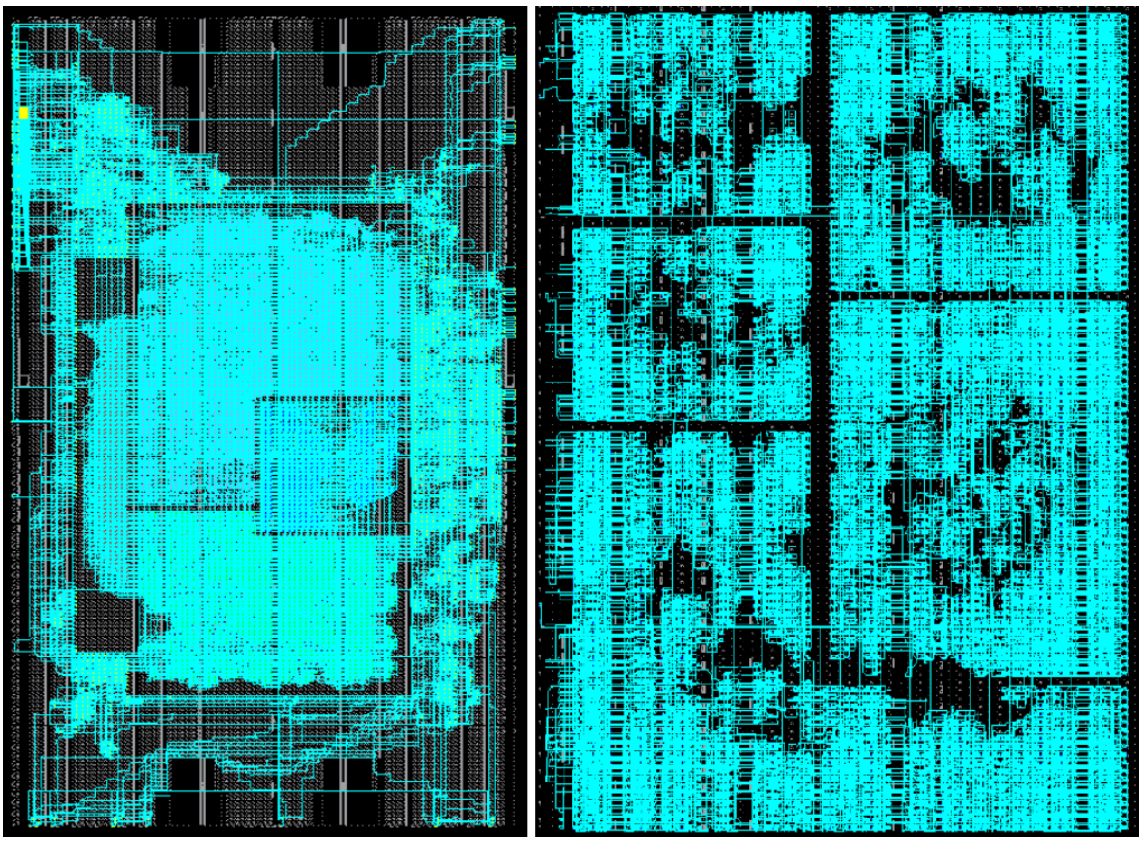


Ilustración 37. IP Cores aislados (7)

Lo que se quiere conseguir con este aislamiento de módulos es poder aplicar una reconfiguración parcial durante la ejecución del sistema. Consiguiendo reservar un espacio en la FPGA lo bastante grande para albergar cualquiera de los IP cores de comunicación nombrados anteriormente y teniendo disponibles los bitstream de cada uno de ellos en una memoria externa, se puede conseguir que el sistema operativo reprograme ese espacio de la FPGA para usar uno de los interfaces de comunicación cada vez que sea necesario sin tenerlos todos implementados a la vez. Esto implica un ahorro de espacio en la FPGA además de una optimización del System on Chip (SoC) ya que se podría llegar a crear un sistema genérico que se auto-reprogramara en uno específico cuando el mismo sistema operativo lo requiera.

10. Bibliografía

1. Wikipedia. [En línea] https://en.wikipedia.org/wiki/DO-254.

2. Wikipedia desarrollo ágil. [En línea] http://es.wikipedia.org/wiki/Desarrollo_%C3%A1gil_de_software.

3. Wikipedia Scrum. [En línea] http://es.wikipedia.org/wiki/Scrum.

4. Imagen Scrum. [En línea] http://epf.eclipse.org/wikis/scrum/Scrum/guidances/supportingmaterials/resources/ScrumLargeLabelled.png.

5. AIM online página 12. [En línea] http://www.aim-online.com/pdf/OVIEW429.PDF.

6. **Xilinx.** [En línea] http://www.xilinx.com/support/documentation/ip_documentation/fifo_generator/v10_0/pg057-fifo-generator.pdf.

7. —. *r2-flow-guidance.*

8. **Martinez, Daniel A.** [En línea] http://www.davi.ws/avionics/TheAvionicsHandbook_Cap_2.pdf.